NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

A LAYERED SOFTWARE ARCHITECTURE FOR HARD REAL TIME (HRT) EMBEDDED SYSTEMS

by

Michael W DaBose

March 2002

Dissertation Supervisor: Luqi

Approved for public release; distribution is unlimited

Report Documentation Page

Report Date 29 Mar 2002	**Report Type** N/A	**Dates Covered (from... to)** -

Title and Subtitle A Layered Software Architecture for Hard Real TIme (HRT) Embedded Systems	**Contract Number**
	Grant Number
	Program Element Number
Author(s) DaBose, Michael	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Naval Postgradaute School Monterey, California	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes The original document contains color images.	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** UU
Number of Pages 145	

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY ***(Leave blank)***

2. REPORT DATE
March 2002

3. REPORT TYPE AND DATES COVERED
Dissertation

4. TITLE AND SUBTITLE: Title (Mix case letters)
A Layered Software Architecture For Hard Real Time (HRT) Embedded Systems

5. FUNDING NUMBERS

6. AUTHOR(S) DaBose Michael

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT ***(maximum 200 words)***

The current state of the art techniques to describe and implement a hard real time embedded software architecture for missile systems range from inadequate to totally non-existent. Most of the existing software implementations within such systems consist of hand coded functionality, optimized for speed, with little or no thought to long term maintainability, and extensibility. Utilizing current state of the art software development technology, the first ever software architecture for hard real time missile software has been designed and successfully demonstrated. This component based layered abstraction pattern approach to software architecture revolutionizes reduced development time, cost, provides an order of magnitude decrease in error, and is the first such software architecture to function within the hard time constraints of the most extreme cases related to missile systems. Additionally, componentization of functionality allows for porting of software developed for one missile to any other missile with no modification. Hardware obsolescence is overcome by software abstraction layers which isolate the hardware instance from the software functionality providing a rapid, low cost transition of software from one instance of missile hardware to another. The end result of this research is a software architecture demonstrating the capability of managing complex functionality in an accurate, quantifiable, and cost effective manner.

14. SUBJECT TERMS Embedded Software Architecture Hard Real Time

15. NUMBER OF PAGES 142

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A LAYERED SOFTWARE ARCHITECTURE FOR HARD REAL TIME (HRT) EMBEDDED SYSTEMS

Michael W DaBose
B.A Miami University, 1978
M.S Naval Postgraduate School, 1997

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2002

Author: Michael W DaBose

Approved by:

Luqi
Professor of Computer Science
Dissertation Supervisor

Valdis Berzins
Professor of Computer Science

Nelson Ludlow
Ph.D. Computer Science

Mantak Shing
Associate Professor of Computer Science

Curt Schleher
Professor of Information Systems

Approved by:
Luqi, Chair, Department of Software Engineering

Approved by:
Carson K. Eoyang, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The current state of the art techniques to describe and implement a hard real time embedded software architecture for missile systems range from inadequate to totally non-existent. Most of the existing software implementations within such systems consist of hand coded functionality, optimized for speed, with little or no thought to long term maintainability, and extensibility. Utilizing current state of the art software development technology, the first ever software architecture for hard real time missile software has been designed and successfully demonstrated. This component based layered abstraction pattern approach to software architecture revolutionizes reduced development time, cost, provides an order of magnitude decrease in error, and is the first such software architecture to function within the hard time constraints of the most extreme cases related to missile systems. Additionally, componentization of functionality allows for porting of software developed for one missile to any other missile with no modification. Hardware obsolescence is overcome by software abstraction layers which isolate the hardware instance from the software functionality providing a rapid, low cost transition of software from one instance of missile hardware to another. The end result of this research is a software architecture demonstrating the capability of managing complex functionality in an accurate, quantifiable, and cost effective manner.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I’d like to thank my daughter Erica and her ever-helpful encouragement in completing this dissertation. A special thanks to Professor Luqi, my mentor, friend, and advisor over the years.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM LOOKING FOR A SOLUTION

Embedded computing is everywhere. From automobiles to the common toaster, embedded processors, and their associated software touch every aspect of our daily lives. As the list of uses for embedded systems grows, along with the associated functionality, software has becoming increasingly more complex. Complexity, coupled with life cycle iterations of the software has produced unstructured, undocumented, and often incomprehensible software source code. The need is for structured and maintainable, software architecture to support hard real time systems, specifically within the missile domain, which adheres to recognized standards of software engineering practices, and lends itself to state of the art development tools and methodologies.

B. ADVANCES TO THE STATE OF THE ART OF SOFTWARE ENGINEERING

A hard real time embedded software architecture is one in which the software execution is reliable, predictable, and executes within the time constraints required by the specific domain (schedulable). While there are other architecture which claim to be real time, Real Time Defense Information Infrastructure Common Operating Environment (RT DII COE), Weapon System Technical Architecture Working Group (WSTAWG), and modern aircraft flight control systems, their domain is related to applications having far too coarse a granularity to support the missile domain. While it is true that such systems are hard real time, within the context of their required operational parameters, the granularity within the flight control system is far too course and slow to support missiles. There is a big difference in response time, and latency tolerances, between a commercial jet flying at 500 knots, and a missile traveling at Mach 2. The advancement of software engineering will be the design principles, and working software architecture, supporting hard real time mission objectives within the missile domain, while maintaining software engineering standards, compatibility with existing software systems, and promote component based reuse, and long term life cycle support. The consequences of such an architecture will provide a controllable, maintainable, and supportable hard real time software architecture, based on layering, and components, promoting true reuse, which in turn will provide a superior product, at lower cost, and delivery within a much shorter period of time than current practices would allow.

C. BACKGROUND

The realm of hard real time (HRT) embedded software can best be characterized as a legacy consisting of on the fly solutions for a very specific problem [25][45]. There are few rules, and little organization as to how things are accomplished. The only concerns have been, and to a large degree continue to be, optimization for speed of execution at all costs, and delivery of the immediate product, without regard to future needs.

A significant problem to be overcome within the HRT embedded software environment is “we have a unique problem”. To a degree this may be true, however, guaranteed scheduling of the true hard time tasks is the same as in other systems of critical execution constraints [25]. What I have observed is that not all the tasks within HRT embedded systems, such as missiles, are in fact hard time. There are aspects of HRT systems, which can be regulated to the soft time domain, alleviating some scheduling difficulties.

A real-time process is where the computation's validity depends on the logical correctness, activity synchronization, and time-sensitive completion. In a real-time system, the time that an activity takes to complete and deliver results is as important to correctness. What is important is not how fast the system responds but that it responds predictably at appropriate times. Hard real-time applies to activities that must be deterministic; critical activities have deadlines. When this processing fails to meet a deadline, the system has failed. In many DOD hard real time systems, failure can equate to loss of life – a missile system is a good example. The design emphasis when building systems with hard deadlines is to guarantee that all deadlines will be met. Soft real-time is non-deterministic to the extent that an occasional missed deadline can be tolerated as acceptable degraded performance, not a system failure. The value of completing a soft real-time activity decreases after its deadline has passed, but the rate at which the value decreases differs between activities.

D. RESEARCH QUESTIONS

The primary purpose of this effort is to devise a practical, working layer abstracted software architecture capable of supporting the domain of hard real time embedded software, while taking into account the ever changing landscape in hardware architectures, and software maturation over time. This leads to the basic research questions to be addressed in this dissertation:

Utilizing current and projected industry standards, how can software be structured to take into account component based functionality?

The inherent complexity associated within today’s software intensive systems required automated relatively error free design and generation tools. The tools should provide the descriptions, and methods allowing precise design and implementation, while maintaining current industry standards. Given that many software intensive systems are quite expensive to build initially, long-term sustainability is of concern.

How can such software architecture abstract the particulars of hardware, and alleviate associated hardware obsolescence?

The specific domain of the subject software architecture is embedded hard real time systems. Processes and associated capability of software has been dictated by hardware speed and scheduling capability. While I do agree that scheduling is more important than speed, without the speed, many capabilities have been beyond the reach of software. It is important that any embedded software architecture takes into account increases in hardware capability, and allow for the utilization, through insertion of new useful technology when feasible.

What realistic design approach is appropriate to provide the responsiveness required for hard real time systems, while taking into account future increases in hardware efficiency, allowing the insertion of more complicated processes in the software architecture?

This dissertation will present a layered architectural approach to abstract specifics of hardware form the actual applications. In addition, a component based application layer will enforce reflective communications, through a middleware layer aiding greatly in tight integration, loose coupling of components, maximizing the opportunity for reuse.

E. GENERAL APPROACH

The design, execution, and implementation of hard real time software intensive systems have only recently been supported by automated methods. I have reached several conclusions based upon my survey of previous work, observations in the working environment, and best judgement as to where the field of such systems is heading. Software complexity on software intensive systems has reached a point of sophistication beyond normal human comprehension. To that end a better way of describing the constructs embodying the software system is necessary. One such methods, the Universal Modeling Language (UML), while not a point of discussion in this dissertation, is seemingly the mainstream of software development, provides the necessary methods and procedures to delineate a software architecture. Coupling the UML model with functional specifics allows automatic code generation of the complete software system. The approach to software design in real time embedded systems requires a more disciplined approach than that currently utilized within the industry. The approach chosen in this dissertation is a layered abstraction method incorporating the best aspects of object oriented programming techniques, while being mindful of the tight scheduling and timing requirements necessary for successful execution of hard real time embedded systems. Finally, the software architecture, which has been developed, and implemented in new generation missile systems, is presented.

The main assumption made in this dissertation is that the effects of Moore’s Law, mainly related to advances in hardware, has reach a level of maturity, and speed to allow software constructs which could not be attempted before within the context of hard real time as related to missile system development.

F. RELIABILITY AND AVAILABILITY

1. A Crisis 50 Years in the Making

According to a growing number of researchers and computer users, the biggest problem facing software engineers is software complexity. "We've known about this problem for 40 years," says Alfred Spector, vice-president at IBM Research. "This is probably the number one problem...It can't go on."

Spector's research shows that IT spending in the United States is nearly doubling every year. Application development is already a costly process: IBM made a large investment on the CICS application server platform, which handles over a trillion dollars / day in transactions [54]. Investments are high, potential returns are higher, and the cost of a mistake can be higher still. A recent example of a minor software mistake is the recent disaster of a French Ariane 5 rocket launch in French Guinea, on June 4, 1996 [55]. A reusable software module, written in Ada, was taken from a smaller, less powerful rocket, and integrated into the Ariane 5 rocket. This piece of software was a part of inertial reference system horizontal velocity. Unfortunately, the legacy software was utilizing a 16-bit field for one of the horizontal velocity parameters. The rapid acceleration of the Ariane 5, coupled with an attempt to convert a 64 bit floating point, to a 16 bit signed integer exceeded the maximum value of the 16 bit representation, causing an overflow condition, failure of the navigation system, and necessitating the destruct of the rocket shortly after launch. Simple mistake, expensive consequence, approximately US$500 million. Software requirements are only increasing availability, security, integration, scalability, and integrity are becoming more complicated issues even as they become more critical. Many problems today require the application of a collection of technologies, not just one. At the same time, the cost to maintain existing software is rising. According to a Standish Group research study [56], only about 16 percent of software projects are completed on time and on budget. The Software Productivity Research consulting firm [56] finds that an average programmer only works about 47 days a year on new development; the other 150 days are spent testing, debugging, or working on projects that get canceled.

G. BRIDGE BUILDING VERSUS COMPUTER PROGRAMMING

The high cost is not due to programmer incompetence. In comparison, building a bridge is something we can touch, see, feel, and have physical limitations, whereas software has only theoretical limits. The design of a bridge simply can't be changed halfway through the project without physically tearing down the work that has already been done. But since the work of a program is invisible, it is deceptively easy to introduce sweeping changes mid-project. Almost always, these changes will greatly complicate the product.

Complexity is part of the essence of software. Software entities are more complex for their size than perhaps any other human construct because no two parts are alike. If they are, we make the two similar parts into a subroutine - open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound. Digital computers are themselves more complex than most things people build: They have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders-of-magnitude more states than computers do. Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger sizes; it is an increase in the number of different elements. In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly [44] [47].

The effort of bridge building is usually on schedule, and budget largely due to its blueprint, a precise technical set of requirements and specifications, around which all planning and development occur. But, given today’s state of technology, it's impossible to create a similar blueprint for a software program. Interestingly, in the history of bridge building, Spector did find overshot estimates, bloated time frames, and even product collapses. But when a bridge fell down, an investigation commenced and the cause of failure was determined. In the software industry, no such audit takes place: More often, software failures are rationalized and ignored. Software producers do not seem to bear the same burden of accountability as manufacturers of physical goods.

Software's inherent complexity can only be managed through careful design. But the world of commercial development tends to work against good design practices. The swiftly changing market pressures developers to rush out a product — any product. An army of programmers creates many large commercial products, each working on a separate piece of the problem. The resulting jigsaw is stitched together any way that works — but without careful, coherent oversight, there's plenty of room for unanticipated interactions in the code.

1. The Failure of ERP

Consider the example of enterprise resource planning (ERP) software, promoted by companies such as SAP and Siebel. ERP software aimed to replace all the separate departmental systems inside a corporation with one central software system. ERP was supposed to make it possible for a sales order to automatically generate the corresponding adjustments in production, inventory, and accounting. Overall the system was incomprehensibly complex. In an ERP implementation project, the ratio of money spent on consulting services to money spent on software is between 5-to-1 and 10-to-1 [33]. Many companies could never get the software to work at all. Just before Halloween 1999, Hershey Food Corp. announced that because of problems with its ERP software, the company was unable to fill orders and get its candy onto shelves. The company reported a 19 percent drop in earnings that quarter. Subsequently, other companies — including Whirlpool and Allied Waste — revealed that they were also facing delays in shipping caused by their new ERP systems. ERP software has often been a failure because it is too complicated to install and use. All too often we solve the hardest problems and don't worry whether anyone can use our solutions.

This is not a problem that will have a single answer. The complexity of ERP software stemmed largely from its attempt to integrate so many different functions. The heterogeneity of the technical landscape is only increasing, and interoperability is critical.

2. Improving the Software Tools

Software must also struggle to keep pace with improvements in hardware. Fifteen years of Moore's law is really amazing. Developers are essentially working in the same ways, with the same tools, to create applications that must be vastly more powerful. Most of the new tools on the market are trying to save people from the fate of having to write code. Really cutting-edge stuff is still done in Notepad, or Epsilon.

Software production won't be comparable to bridge production until the design process is similarly firm, and if there is hope for software design it will lie not in the commercial sector but in computer-science research. Product-oriented programmers, subject to extreme time constraints and the pressures of an ever-changing market, will invent clever solutions to their dilemmas; but clever workarounds are part of the problem.

3. Concurrency

To respond effectively to events that occur asynchronously in the environment, with which the system interacts, real-time systems are often constructed as collections of concurrently executing tasks and processes. In contrast with the concurrent processing inherent in general purpose computing, where processes compete for resources without interacting in other ways, the tasks and processes of real-time systems cooperate closely to achieve mission objectives [22] [23] [27].

4. Operational Environments

Real-time systems often must operate in extreme environments that are far less accommodating than a typical office or computer facility. These systems are often installed on vehicle platforms, e.g., aircraft, tank, or missile. Environmental conditions can be expected to exert significant space, weight, and power consumption constraints. Also, the hardware often must be designed to withstand environmental stresses such as extremes of temperature, shock, vibration, corrosive atmospheres, poorly conditioned electrical power, and severe electromagnetic fields. In addition, many of these systems have long shelf lives, during which said systems must be maintained. The obsolescence of hardware, and future non-availability must be considered in the design and migration to meet future maintenance requirements. These considerations significantly restrict the choice of equipment that can be packaged with the real-time system.

The current state of the art for hard time real time systems development relies upon developer knowledge, gut instinct, a feeling of simplicity, and beauty in the design principles employed in system software design and development. Unfortunately, while such design approaches can solve very specific problems of a particular system, they do not lend themselves to adaptability to other related systems with respect to code reuse, standardized software architecture approaches, system growth, or maintainability.

A significant aspect of HRT embedded software is the software architecture itself. Efforts such as Lockheed Martin’s HardPack, and the Real Time Defense Information Infrastructure Common Operating Environment (RT DII COE) [30] [37] [39], have not been successfully demonstrated within the HRT domain. It is my contention that a fundamental flaw exists within each of these approaches. In each case, the design, and related overhead associated with a soft real time domain are being applied to HRT software. I believe that the design of a software architecture, or family of architectures, must reflect the necessities of the environment within which they exist.

For example, consider the RT DII COE. The layered abstractions within the architecture include everything from database capabilities to protocols.

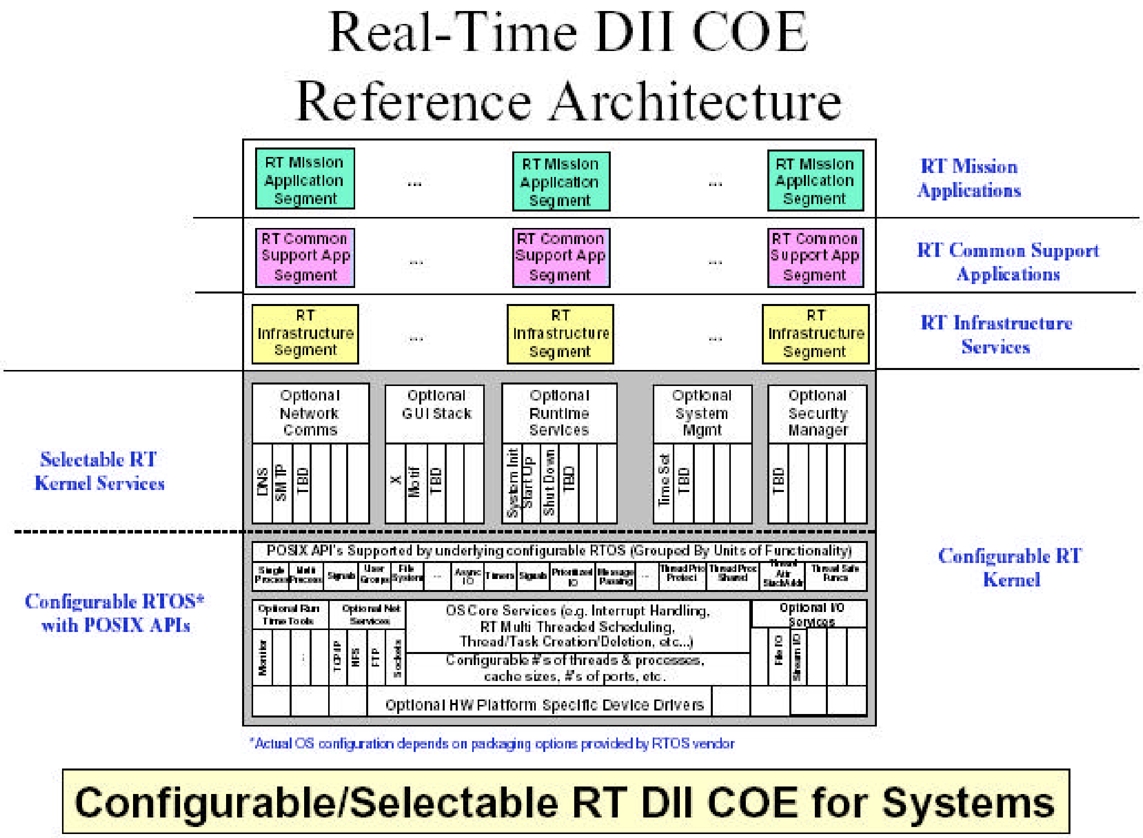

Real-Time DII COE
Reference Architecture
RT Mission Application Segment
...
RT Mission Application Segment
...
RT Mission Application Segment
RT Mission Applications
RT Common Support App Segment
...
RT Common Support App Segment
...
RT Common Support App Segment
RT Common Support Applications
RT Infrastructure Segment
...
RT Infrastructure Segment
...
RT Infrastructure Segment
RT Infrastructure Services
Selectable RT Kernel Services
Optional Network Comms
DNS
SMTP
TBD
Optional GUI Stack
X
Motif
TBD
Optional Runtime Services
System Init
Start Up
Shut Down
TBD
Optional System Mgmt
Time Set
TBD
Optional Security Manager
TBD
POSIX API's Supported by underlying configurable RTOS (Grouped By Units of Functionality)
Single Process
Multi Process
Signals
User Groups
File System
...
Async IO
Timers
Signals
Prioritized IO
Message Passing
...
Thread Prio Protect
Thread Proc Shared
Thread Attr StackAddr
Thread Safe Funcs
Configurable RTOS* with POSIX APIs
Configurable RT Kernel
Optional Run Time Tools
Monitor
...
Optional Net Services
TCP/IP
NFS
FTP
Sockets
OS Core Services (e.g. Interrupt Handling, RT Multi Threaded Scheduling, Thread/Task Creation/Deletion, etc...)
Configurable #'s of threads & processes, cache sizes, #'s of ports, etc.
Optional I/O Services
File I/O
Stream I/O
Optional HW Platform Specific Device Drivers
*Actual OS configuration depends on packaging options provided by RTOS vendor
Configurable/Selectable RT DII COE for Systems

Figure 1. Real Time Defense Information Infrastructure Common Operating Environment [37]

While such an environment might be suitable for a Command and Control System, which is not hard real time, the overhead latencies introduced by such a structure are not acceptable within the context of hard real time, as pertaining to a missile system. Most embedded systems deal with limited resources, both in terms of processing power, and storage capability (RAM and permanent). While the increased miniaturization, couple with hardware speed increases has alleviated this situation, to a degree; resource allocation is still a major problem within such systems. To produce the layered abstraction desired within a missile system, while maintaining the requisite performance requirements necessitates a more selective approach in the missile system design. Much of the capability within RT DII COE is not required (yet) within the missile, and hence such architecture is not suitable for missile applications. The notion of one design fits all is not likely to be a reality any time soon.

The understanding of such an environment is key to successful implementation. In all fairness, it must be noted that the RT DII COE Working Group notes that the initial application of their architecture is directed toward soft real time systems of the C2 – C4I realm. [37]

5. What of the Future?

Without better tools and methods, programmers on a tight deadline are actually less likely to incorporate preexisting components into their work. The component will probably need to be modified anyway, and the time it takes to locate, understand, and adjust the code can't be justified in the short term. And of course such programmers are even less likely to create a reusable component because that would triple the workload. Unless or until researchers come up with vastly improved developer tools or a fundamental breakthrough in software design, developers can only attempt to control the rising tide of complexity.

H. CONTRIBUTIONS

The first contribution of this dissertation is the association of current software technology standards toward the solving of the hard real time software architecture constraints such as componentization, layered abstraction, and time synchronization. Although the toolset for such a design and implementation is not complete, or mature, it is possible to address this problem successfully, while keeping the design open to allow insertion of new technology and methods in the future. Assessment of this technology and the analysis of various architectural approaches are covered in chapter II, sections A through C.

A second and most important contribution is the development of a functional hard real time embedded missile software architecture. Through utilizing examples of soft real time architectures, such as the Defense Information Infrastructure Common Operating Environment (DII COE), and others, the necessary functionality for a missile system, comprising a well understood domain [62], has been transposed to a layered component based approach, entailing the necessary services, and not the unnecessary overhead associated with soft real time domain. The proposed architectural design and actual implementation in a real missile is discussed in chapters III and IV, with the actual implementation in chapter V.

The third contribution of this dissertation is the realization that the benefits related to Moore's Law, processing capability doubling every eighteen months, or less, can be utilized to advantage when challenging previously unthinkable problems. Embedded processing solutions can address a number of areas of research ranging from communications, to weapons systems, to medical technology. We have the know how as to what needs to be accomplished, but until very recently did not have the computational horsepower to effectively implement the solution in a cost effective manner. The third contribution, while not directly discussed, is one of realization that current processor technology allows the re-examination of problems, in light of new enhanced capability. What was considered not feasible in the past, may if fact be quite feasible today.

II. ASSESSMENT OF PREVIOUS WORK

A. CURRENT TRENDS IN HARD REAL-TIME SYSTEMS

1. A World in Transition

The world of intelligent devices is changing dramatically. The computerized devices around us are getting smarter, they're increasingly connected and interdependent, and they're becoming vastly more numerous. And all this is happening at an ever-increasing rate.

Blame it on Moore's Law, but it's now practical to embed moderately high performance computing and connectivity in just about everything that runs on electricity -- whether tethered or mobile. This trend is fueled by powerful and highly integrated system-on-chip processors, coupled with large capacity system and storage memories (both disk and silicon), and empowered by wired and wireless communications interfaces (Ethernet,IrDA,802.11,BlueTooth).

Another important phenomenon is that as both embedded computing and connectivity proliferate, the intelligence within tomorrow's devices is becoming less localized. Increasingly, the precise location of the software running on these devices is becoming less visible, and eventually we're unlikely to actually know where the programs we use are located.

2. Welcome to the Real World

As the boundaries of the traditional computing paradigm blur and a new reality based on distributed, interconnected, pervasive computing devices dawns, a few important attributes of the coming era draw into focus . . .

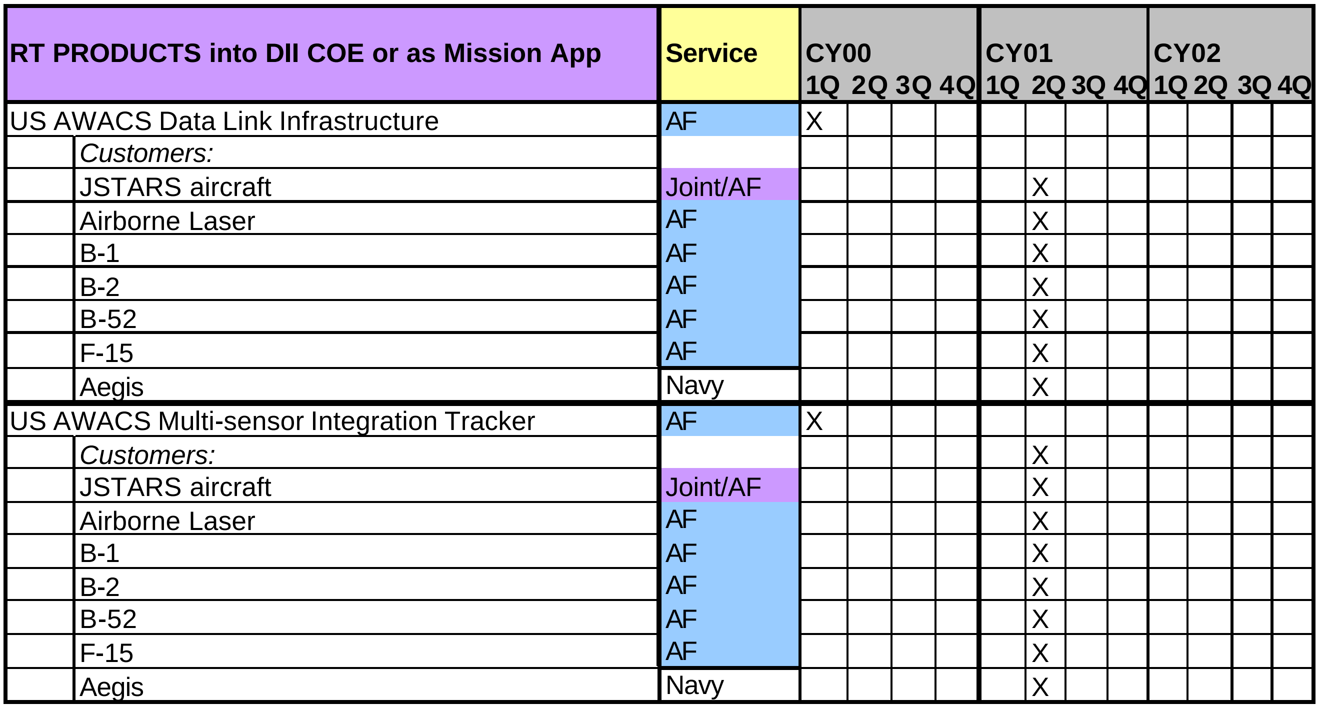
- The number of smart devices (i.e. products with embedded operating systems inside) will grow exponentially, reaching numbers in the billions.
- The choice of CPU will be more a matter of cost than technology or architecture.
- Nearly all devices will have connectivity, whether wired or wireless.
- The system, vice the singular computer will become the computational platform. Definitions of what is the "computer" will be hard to describe.
- Most devices will have specific rather than general-purpose functionality, so their application software will be defined by their manufacturers (rather than loaded by their users). I'm not sure if this is a good thing.

3. And what about alternatives?

The price of portability is overhead, and slower system performance. In comparison of the current possible implementation methods for hard real time systems, a common latency penalty of 30%, or more is realized between the same applications implemented in the traditional "hardwired approach" (intimately tied to hardware, non-portably), and portability methods including Message Passing Interface, Real Time Defense Information Infrastructure Common Operating Environment (RT DII COE), and Weapons Systems Technical Architecture Working Group (WSTAWG) Common Operating Environment [64]. In observing each of the above methods, none were found suitable for the following reasons:

Message Passing Interface (MPI) – While showing a promise of potential within inter processor communications in a heterogeneous environment, the lack of maturity, and associated slowness of the available academic MPI source codes (version 0.9). Commercial versions of MPI are available through MPI SoftTech, but the implementations targeted are main stream embedded processor, on standardize COT boards, making utilization of a commercial vendor not economically feasible. While it might be possible to advance the academic version of MPI to usable implementations within missile, this would require extensive investments in time and money. The organization I represent is a consumer of software such as MPI, not a developer. In addition, any enhancement made to the academic version of MPI, outside the standards organizations, would be a unique singularity, requiring substantial upkeep. Finally, the MPI standard seems to be fading in popularity and support. Newer, emerging standards, while have yet to be realized, RAPIDIO, and INFINIBAND, supported by opposing commercial interests will replace MPI at some future point. Investment in MPI, like Ada, would not be a wise choice due to questions of long term support. Missile products commonly have a shelf life of 20 years, or more.

RT DII COE – Real Time DII COE, as the name implies is an extension of the DII COE, while is tailored to the needs of soft real time systems, such as those found in C4I(SR), digital communications, and so on. In keeping with the component-based philosophy of DII COE, RT DII COE maintains much of the functionality, and associated overhead found within RT DII COE, see figure 1. Based upon runtime data provided at the RT DII COE outbrief, held in Chantilly, Virginia, in December 2000, RT DII COE test results are worse than MPI benchmark data performed in my laboratory, making RT DII COE not worth consideration.



RT PRODUCTS into DII COE or as Mission App		Service	CY00				CY01				CY02			
			1Q	2Q	3Q	4Q	1Q	2Q	3Q	4Q	1Q	2Q	3Q	4Q
US AWACS Data Link Infrastructure		AF	X											
	Customers:													
	JSTARS aircraft	Joint/AF						X						
	Airborne Laser	AF						X						
	B-1	AF						X						
	B-2	AF						X						
	B-52	AF						X						
	F-15	AF						X						
	Aegis	Navy						X						
US AWACS Multi-sensor Integration Tracker		AF	X											
	Customers:							X						
	JSTARS aircraft	Joint/AF						X						
	Airborne Laser	AF						X						
	B-1	AF						X						
	B-2	AF						X						
	B-52	AF						X						
	F-15	AF						X						
	Aegis	Navy						X						

Figure 2. RT DII COE Target Platforms

As noted in figure 2, RT DII COE target platform integration, while comprising real time utility, does not speak to implementation in hard real time systems. In each case, when processing deadlines are missed, the result is degradation, not catastrophic failure. The operating system (OS), to which RT DII COE is specifically tailored if LYNX, which is not a real time operating system, and does not lend well to porting the OS to new hardware instantiations. RT DII COE is not hard time with respect to missile systems, and can not meet the granularity requirements of missile systems.

Weapons Systems Technical Architecture Working Group (WSTAWG) Common Operating Environment (WSTAWG COE) – In the fall of 1999 I met with Bennett Scott, of Raytheon Technical services, in Spring Creek, Texas, then the leading developer, and integrator of the WSTAWG COE. A three-day evaluation of this system was conducted in concert with the development team. The following conclusions were reached:

1) WSTAWG COE is currently operating in a soft real time environment, mainly the Bradley Fighting Vehicle.
2) The COE is designed to support soft real time requirements, not hard real time.
3) The WSTAWG COE is tightly coupled to the hardware of the Bradley Fighting Vehicle, and can not be ported to new hardware instances without significant modifications. The level of effort required would at least be equal to developing a new approach with respect to software architecture.
4) The WSTAWG COE supports Ada83, Ada95, and C bindings. The environment does not recognized or support object oriented development methods.
5) The WSTAWG COE is a hand crafted development effort with little or no software development automation support. Any development or support of such a system would be labor intensive, and error prone due to the lack of automation, prototype simulation validation, and associated error checking capabilities.

The end analysis was the WSTAWG COE would not support the objectives and goals of missile hard real time software architecture without extensive modification at least equaling the cost of developing missile software architecture from scratch.

To date, there are no known software architectures in existence, or planned to be in existence, within the foreseeable future which support the requirements of missile systems.

As the number and variety of devices with embedded intelligence grow exponentially, the need to minimize cost and maximize specialization increases correspondingly. Hence, Embedded Linux becomes a highly desirable technology for the operating system due to its scalability, configurability, and affordability.

It's worth noting that until recently, the "cost penalty" associated with the CPU and memory resources necessary to run Linux, actually a flavor of the Linux RT variant, had been a somewhat limiting factor relative to using it in cost sensitive devices. Now, however, the baseline needs of embedding Linux -- roughly 2M Flash and 4M RAM memory and a moderate speed processor (100+ MIPS) -- have become reasonably inexpensive, thanks in large measure to Moore's Law. While Linux offers potential in the future, it is currently not considered a contender to the operating system within missiles due to lack of maturity and stability with respect to standards. As Linux matures, it may very well be the OS of choice. The current design approach for the missile software architecture is non-OS specific.

4. Can Java help?

Another important challenge in this new era in which we'll be surrounded by billions of increasingly intelligent devices, all communicating with each other, is the obvious need to simplify and quicken the process of application development, deployment, and maintenance. In this regard, Java appears postured to play an increasingly important role. While I believe that RT Java still has a way to go before practical implementation in hard real time systems will be feasible, due to current latency issues; Java offers a compelling option for the future.

Although Java failed to hit the target for which it was initially developed, which was, ironically, to serve as an embedded operating system within smart devices, Java ended up providing a practical means to enable moving applications around among computing devices.

Today, despite its early failure as an embedded operating system, Java is showing promise in the role of providing a device-independent application platform, running on top of the embedded operating system. In this case, rather than serving as the operating system itself, Java provides the benefit of masking the unique aspects of the underlying device and providing an array of services beyond those offered by the embedded OS.

In the context of an exponential proliferation of smart devices, Java is emerging as a handy way to minimize device-specific development and allow developers to focus on the truly unique aspects of their projects. Increasingly, Java is providing a means to obtain functionality like GUIs, Web browsers, protocol stacks, handwriting and speech recognition, wireless communications, multimedia support, database management, and a wide range of remote services.

Recent events involving Java place reasonable doubt as to the future supportability of Java. The Microsoft .NET initiative, coupled with the innovative approach of SOAP, and XML based application / information exchange mechanism, highlight that a course of action, utilizing a Java-like approach, build once, run everywhere, is far from stable. The risk is deemed too great at this point in time to consider Java. Perhaps in the future, once standards stabilize, Java, or another such approach might be appropriate.

5. The Growth of Software Complexity

Software is inevitably growing in complexity, for several reasons:

- Business computing for average throughput and scientific, industrial computing for real-time device control are converging.
- Functions are migrating from hardware to software as computers take over electromechanical devices.
- Software is growing more complex in and of itself.
- There is increasing interaction between real-time and non real-time systems. Not all aspects of real time software have hard deadlines.
- Customer demand is increasing. Such demand will drive technology investments, and resulting application of new technology.

There is a crisis now in software development in the embedded systems space; it is getting difficult to produce software. This is a problem relating not to time, money, and people, but complexity, and an unwillingness to throw away troublesome legacy code.

In the 1950s, when IBM divided computing into two branches, -- business computing for average throughput, and scientific computing for real-time device control, the IBM architects decided that these two branches, which called for differing optimizations, should remain split. As more applications, not only scientific, but everyday use are demanding the performance of real time, it is coming to the point now where those two branches are going to collide, and we are going to have to merge them back together.

As computers become increasingly ubiquitous in daily life, they begin to take control of what were once electromechanical devices. The demands for software now are different from traditional business computing, where you are just manipulating data; now, you are actually controlling the real world.

The networking of computer devices is increasing their complexity and flexibility, and software is taking over functions previously assigned to hardware. Together this leads to ever more complicated software.

Complexity grows radically with the size of the software system. To implement increased, and more complex functionality in embedded systems, more and more of it has to be in the software.

There is a dark side to Moore's law -- the one that states the CPU power of a processor doubles every 18 to 24 months. The dark side of this is the other theorem that says software grows to fill the available hardware. So if the hardware is growing exponentially, the software has to bulk up to fit the space.

Although only a small part of most sophisticated software systems are actually real-time, real-time software adds substantially to the complexity of the system. The pieces that actually need to be real-time tend to be small, but if the entire system has to be programmed under the same kind of rules the real-time system uses, you radically increase the complexity of the system.

6. The Merging of Real-Time and Non Real-Time Systems

There will be increasing interaction between real-time and non real-time systems as the non real-time functions on devices interact with the real-time parts of the device. The key to real-time systems will lie in the response time of the system. Real-time systems have a narrow response time distribution relative to their domain, while non real-time systems have a wider distribution, the goal is to be able to get these two systems together and give the programmer the ability to program in these systems cleanly.

Real-time is not necessarily "real fast" according to Gregory Bollella [27]: "It is useful to look at the relationship between a control strategy and the number of objects that have to participate in implementing that control strategy. At a device level, you have a control strategy with some code that controls the device, and typically you have a hundredth of a second for ten objects to participate in the strategy. Now, control strategies are envisioned which span a whole enterprise and require the participation of hundreds of thousands of objects and require response times measured in the hundredths of seconds. Both strategies require real-time programming. We are seeing, in automation, that control strategy no longer is just about devices.”

A further factor increasing complexity is customer demand, with customers in a variety of domains wanting products to do more, faster. There is a race to drive function into devices, leading to an inevitable increase in software complexity.

7. The Future of Java Technology in Embedded Systems

Even though Java technology is already in consumer electronic devices, mobile phones, PDAs, and set-top-boxes, the next move is to system-software. Java technology must move from just applications in devices down into system-software. The isolation of Java applications, currently implemented through a Java Virtual Machine, introduces latencies that are not acceptable within the missile domain. As processing speeds continue to increase, and other initiative resulting in Java, native code compiler [58], it might be possible to utilize Java at some future point in time.

According to Gregory Bollella [27], in designing a specification, a set of guiding principles is utilized to make the trade-off that you have to make during the design phase. A system must support the execution of logic that was predictable with respect to time. That is a number one goal. This is accomplished by taking sources of unpredictability out of the Java runtime. The Java Specification is backward compatible, meaning that if you implement the specification in any particular Java edition, programs that ran on that edition previously will continue to run here.

B. EMERGING TECHNOLOGY FOR FUTURE CONSIDERATION

1. JBED: Java for Real-Time Systems

Jbed, Java for embedded system, is a small, fast Java Virtual Machine (JVM) for embedded real-time systems [58]. It also includes a complete real-time operating system (RTOS). In designing Jbed, the first goal was to make a seamless combination of an RTOS and JVM without any glue code in between. Jbed uses either a precompiled native image, or a built-in compiler that translates Java bytecode into native code at load time. The Jbed native run time code executes up to 50 times faster than a Java VM implementation in a given hardware environment [58]. This technique will be referred to as Just Intime Compilation (JIC).

Because Jbed always runs natively, it is a very fast Java system. That it is also a small system is surprising, but leaving out the interpreter and not relying on other processing intensive techniques saves memory. Additional memory saving is achieved by writing everything in Java -- even the device drivers (to write device drivers in Java, Jbed provides a built-in class with methods for direct memory access). Of course, you can link to legacy code written in C, but if you refrain from that, only the Java memory management has to be loaded, and a lot of overhead is avoided. A complete Jbed system with JIC is only about 256 KB and uses about the same amount of RAM to run. A system with TCP/IP and a web server is about 128 KB. Minimal precompiled systems without the JIC can be smaller than 10 KB. These numbers include both the RTOS and JVM.

JIC technology has another benefit. Code loaded dynamically into the running system is compiled immediately, then executed at full native speed. This makes Jbed ideal for applications where interpreters are too slow, but where hot-loads of patches or new applications are necessary.

A final design goal was to make Jbed suitable for hard real-time applications. Hard real-time usually means that no outside event must ever be handled too late. More precisely, each event has a given deadline and missing any such deadline results in an error condition (soft real-time, on the other hand, allows for occasional deadline misses). Real-time applications are more challenging under a Java-based operating system than under other systems because Java uses garbage collection. The garbage collector used in Sun's original JVM cannot be interrupted and is therefore unsuitable for real-time applications. Instead, an incremental garbage collection algorithm has to be used that can be interrupted anytime.

Jbed is an open source design, and is therefor not dependent on a single vendor. It comes with its own integrated development environment (IDE), or can be used with Symantec's Visual Café, IBM's VisualAge, and the like. Jbed runs on Motorola 68xxx and PowerPC processors, with Intel, ARM, and MIPS processors support coming soon.

2. Event-Triggered versus Time-Triggered Systems

In event-triggered systems (based on interrupts) that use multiple sensors (heat sensors, oil-pressure sensors, sensors triggered once per rotation of a moving part, etc.), each trigger is an interrupt handled by an interrupt routine. The interrupt routines are short, and as much work as possible is delegated to nonreal-time threads. (Here, we use the term "thread" exclusively for Java style, nonreal-time threads, and we use "task" for real-time tasks.) Each interrupt routine has a priority. Routines with higher priority will interrupt routines with lower priority.

In such a system, priorities have to be assigned (possibly dynamically) such that each external event is serviced within its deadline. For instance, the interrupt routine that must be called once per rotation of a moving part must have a priority that guarantees it is still called once per rotation even under special circumstances (such as a malfunction of another part of the machine). If this is not guaranteed, a malfunction could spread quickly. In a complex system, this priority assignment can be very difficult to debug.

In time-triggered (deadline-driven) systems, on the other hand, all sensors are sampled (polled) at regular intervals. Each sensor causes the execution of a small piece of code, similar to an interrupt routine. Again, as much of the calculation as possible is delegated to nonreal-time threads. The routines triggered by the sensors have deadlines that are usually known a priori and do not need special priority assignment. For example, a rotating part that rotates a maximum of 1000 times per second has a natural deadline of 1 millisecond. The scheduling is based on which deadline is the most urgent (earliest deadline gets first scheduling).

Because the routines usually do not perform complex calculations, it is possible to assign them duration (that is, a maximum execution time). Again, the duration is known *a* *priori* and, therefore, the scheduler can do a load-time analysis of the schedule of the system under all circumstances. The requirement to have a correct estimate of the duration of the real-time tasks is not a serious problem in practice, because well-designed tasks are simple and the estimates can be on the safe side (and the actual durations can be obtained by other means).

How does the responsiveness of event-triggered systems compare to their time-triggered counterparts? Under small loads (few external events), event-triggered systems are more responsive because each event is serviced immediately (unless a higher priority event comes in between). Time-triggered systems have a response time given by the sampling period. Under high loads, however, time-triggered systems excel because they have much less overhead than event-triggered systems [27].

Time-triggered systems have another advantage. If the machine is extended by another rotating part, it is easier to fit new control software into the system if it is time triggered. The service routine of the new part has a known deadline and duration, and can therefore be added to the system without changing the existing service routines. If the system can still be scheduled, no further work has to be done. Event-triggered systems, however, will likely require many changes. Time-triggered systems, thus, pave the road for the use of reusable software components in real-time systems.

C. THE NEAR TERM OUTLOOK

After examination of available possible candidates for hard real time software architecture, supporting component based abstractions, through a layered pattern, the conclusion is that there are no viable alternatives to designing, and implementing such architecture from scratch. The following examination of components will lead to a realistic approach toward the definition of such an architecture, and answer the research question – Utilizing current and projected industry standards, how can software be structured to take into account component based functionality. In addition, a contribution of this dissertation to the advancement of technology will demonstrate how the proper association of current software technology standards, can be incorporated into a sustainable, layered, component based software architecture capable of meeting the hard real time requirements of missile systems.

1. What is a component?

According to the "OMG Modeling Language Specification" (Revision 1.3): a component is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of a system's implementation, including software code (source, binary or executable) or equivalents, such as scripts or command files. As such, a component may itself conform to and provide the realization of a set of interfaces, which represent services implemented by the elements resident in the component. These services define behavior offered by instances of the component as a whole to other client component instances.

In the UML, a component is a classifier, and so may realize interfaces, have executable behavior and state. It also aggregates other model elements, and may reside on a node (processor) during execution. These things are also true of subsystems. The UML defines a subsystem as a "special kind of Package that represents a behavioral unit in the physical system, and hence in the model" ("OMG Modeling Language Specification," Revision 1.3). A subsystem is a metasubtype of both classifier (same as a component) and package (different than component). However, packages are distinguished by the fact that they can contain model elements, so we are left questioning how subsystems and components differ [28][30].

Systems can contain subsystems, which can contain components, which can contain objects.

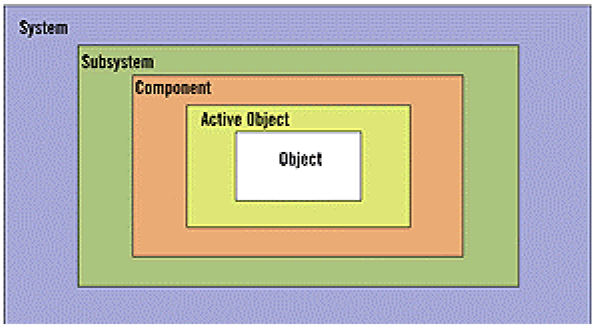


Figure 3. Architectural Elements by Scope

While Figure 3 reveals where components typically reside in the taxonomy of architectural things, it doesn't explain what they are and how they are used. The following criteria are offered as a baseline to define the functionality of components [25][28][30]:

- May be used by other software elements (clients).
- Clients without the intervention of the component's developers may be used.
- Includes a specification of all dependencies (hardware and software platform, versions, and other components).
- Includes a precise specification of the functionalities offered.
- Is usable on the sole basis of that specification.
- Is composable with other components.
- Can be integrated into a system quickly and smoothly.

However, this definition applies to objects of all sizes, because a component is really a large-scale object used in a specialized way. Clients without the intervention of the components' developers (if they are documented) can use objects, components, subsystems and even systems. In fact, objects, components, subsystems and systems should specify the interfaces, system-visible dependencies and functionality. Components must also fit together into collaborations to realize larger scale behavior and ought to be easily integratable.

2. Component Behavior

Much has been said about component behavior [30][59][60]. One line of thought views the components interface as a detailed set of services that can be invoked by its clients. This appears to be a reasonable view. However, the names of the operations and their parameter lists are hardly an adequate description to understand how the interface should be used, what services may be invoked under different circumstances, when one service might be more appropriate than another, and so on.

A somewhat more abstract way to think about the interface of a software architectural unit, such as a component, is the use case. A use case is a named capability of a system that does not reveal or imply the implementation of the capability–in this sense, it's a high-level view of an opaque interface. However, the use case also includes the operations, their signatures, pre- and postconditions, and the set of allowable sequences (the protocol) of the interface [28][29][30].

We can use the notion of a use case as a high-level view of an interface to help us capture the aspects of the component behavior needed for application programmers to determine how to use the component.

As a simple example, let's consider a traffic control system which contains a number of components: several instances of a traffic light component, a control component which manages the interaction of the traffic light components, a communication component that allows them to communicate, and so on. The traffic light component might have a small set of use cases, as shown in Figure 3.

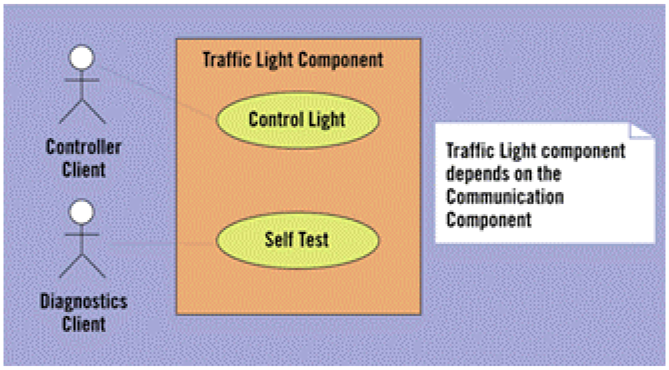


Figure 4. Traffic Light Component Use Cases

Figure 4 illustrates the interface of the traffic light component at a high level. This use case diagram serves to highlight the major capabilities of the components. This aspect wouldn’t be included in the more detailed view offered by merely listing the operations provided by the component. Higher level abstractions of a system serve to provide a “big picture”, and keep detailed design more objective to the overall system goals.

3. State and Components

In general it is useful to think about behavior as being in one of three different categories: simple, continuous or state.

Simple behavior does not depend upon the component’s history. Continuous behavior depends on the history of the object but does so in a nondiscrete way. Amplifiers, fuzzy logic and neural networks all exhibit behavior that acts in this way. The output depends on the current input and its history.

State behavior similarly depends on history, but in a discrete way. An object, when in state X, accepts a particular set of input events, does a particular set of actions and activities, and can reach a particular set of subsequent states. Different states differ in one or more of these three properties. So a sensor, when in the "waiting for data" state, might accept the "data ready" event, perform the "poll for data" action and go to the "has data" state when the "date ready" event occurs. When in the "has data" state, a sensor might then accept the "request for data" event, perform the "filter data" activity and return to "waiting for data" once the data has been read by the client.

Executables are defined by behavior. They can have any or all of the three categories of behavior (even at the same time) in different behavioral aspects. Components, being executable, therefore may have state. A component has state behavior when its behavior depends on its history in a discrete way. One can imagine a component for a traffic light controller that has states of red, yellow, green, flashing red, flashing yellow and even flashing green.

4. Interfaces and Dependencies

The signature of an operation is only one of three crucial aspects of its interface; the other aspects are preconditions and postconditions.

In order to facilitate replaceabilty, components typically provide strong encapsulation and opaque interfaces.

When the services offered by a component interface can be called at anytime, the component (externally) exhibits simple behavior. When some operations either may not be invoked or act differently at different times, then the interface is either continuous or state-driven. It’s often useful to capture the state behavior of an interface via a statechart. In fact, this is commonly called the protocol of the interface. Formally speaking, the interface protocol is a subset of the preconditions, but practically speaking, it is a very important subset.

Consider our traffic light control component and its Control Light use case. There are a number of rules that govern its behavior:

• The component’s available conditions of existence (states) are flashing yellow, flashing red, red, yellow and green.

• Once the component accepts an event to change from red to green, the transition shall take two seconds.

• Once the component accepts an event to change from green to red, it shall wait three seconds, turn yellow for two more seconds and then turn red.

• Once initialized, the component shall always accept a command to go into a "flashing red" or "flashing yellow" state.

• The component can go immediately from the "flashing red" state to a "red" state when it receives an event to do so.

• From the "flashing yellow" state, the system shall turn yellow for two seconds and then turn red when a "turn red" event is received

This component is expected, by its client, to fulfill the contract specified by these rules. We can capture these rules using a statechart as shown in Figure 5. Notice that the "turn green" event is ignored in all but the "red" state (in other words, the component is specifically expected to quietly discard the event if received while in any other state). This statechart specifies the expected externally visible behavior of the interface.

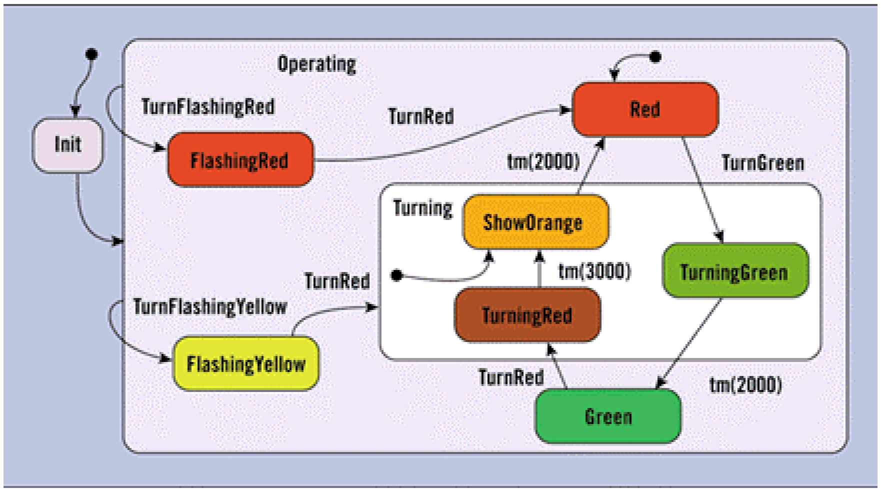


Figure 5. Traffic Light Component States

An even more detailed view of the interface than the statechart is the individual operation [30]. An interface is a named set of operations as well as the protocol that describes their usage. This detailed view can be provided in a number of ways. It is common to merely list the entire set of operations, along with preconditions, postconditions, parameters and their types, return values and the means necessary to invoke the operations, commonly by calling a member function.

Another approach is to show a set of objects contained within the component that are accessible to the outside. These interface or boundary objects provide a way of packaging the operations into smaller coherent bundles [28][59]. They can be shown on a class diagram representing the component while the rest of the objects inside the component remain hidden.

Components don’t often stand-alone. They are frequently arranged so that more abstract or client-friendly services can be offered, and the implementation uses more concrete or less client-friendly components to realize that behavior. The reason dependencies must be identified is because these more concrete services are not solely encapsulated within the component. Therefore, replacing one component with a new revision might involve replacing one of the components on which it depends.

Objects must do exactly the same thing when they collaborate. Objects collaborate to realize large-scale behaviors, and they can do this at various levels of abstraction. At the highest level of abstraction, the collaborating objects are the "system" and the actors that interact with it. The system object provides capabilities, identified through its use cases. The next level down is the subsystem in which large-scale architectural pieces of the system collaborate to realize the system use cases.

5. In Conclusion, a Component is . . .

A large-scale executable object designed to be opaque, with well-defined interfaces, and easily replaceable as a unit in the context of a system [28].

A component’s other aspects exist merely to facilitate that purpose. Opaque interfaces help replaceability because they allow different components to realize a behavior differently, as long as the interface is maintained. Behavior of the component interface can be arbitrarily elaborate and specified using various means for modeling behavior. A component is large scale because it contains (via composition) smaller objects (aggregation, sub-components, or other structures), some of which may be active in the UML sense (that is, they form the roots of threads) [28][29].

The primary use or purpose of components is to facilitate construction of reusable pieces that can easily be inserted into a wide variety of systems without requiring knowledge of the internal implementation of the component, and within the context of a system, facilitate the evolution of that system with specialized, extended, elaborated or optimized versions of components that realize the same set of interfaces.

Components are special kinds of objects that designed to facilitate the realization of a particular aspect of their quality of service–replaceability. Because of this, components have opaque interfaces and extensively document the interface's details and dependency on other components. The principle benefit is being able to construct systems in a manner similar to the process electrical engineers follow today–by piecing together different large-scale pieces, enabling systems to become easier to maintain, evolve and even construct.

An examination of current and evolving methods reveals that a component based approach is the most logical course of action to pursue with respect to the overall software architecture. The component provides a building block for the management of complex software systems. This partially answers the research quest, utilizing current and projected industry standards, how can software be structured to take into account component based functionality? Components effectively isolate dependencies and promote a modular implementation approach enabling true component-based reuse. The component approach is only one piece of the puzzle. Next, the component approach will be related to the supporting infrastructure enabling the realization of the entire software architecture for hard real time embedded systems within the missile domain.

THIS PAGE INTENTIONALLY LEFT BLANK

III. A PROPOSED APPROACH

A. A THIN LAYER HARD REAL TIME SOFTWARE ARCHITECTURE

1. Middleware General Concepts and the Hard Real Time Domain

First we define certain terminology relative to software engineering;

Programmer – a vocation acquired either by on the job training, or more formal study. This is the traditional realm of computer science. Programmers code software to achieve the required functionality of a software system. Most software engineers, myself included, originate from this background.

Software System – A conglomerate of related software “programs” bound together into a comprehensive software package comprising the functional portion of a system. Software is the “brains” of today’s complex systems. More specifically, the software system is the orchestrated manifestation of the systems software requirements. Today’s software intensive systems, which includes just about everything we use in life, are becoming exponentially more complex, over time. Software complexity, in large systems, has reached a point of being beyond any single human understanding, hence the discipline of the software engineer – to bring order to chaos.

Software Engineer – A professional and academically recognized engineering discipline dealing specifically with the design, implementation, and life cycle support of complex software systems. Some, but not all, of the issues dealt with by software engineers include design, development methodology, technology insertion, invention, risk mitigation, cost avoidance, integration, and verification.

Software Architecture - An architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition.

The software architecture of a program or computing system is the structure or structures of the system, which comprise slots for software components, the externally visible properties of those components, and the relationships among them. By "externally visible" properties, I am referring to those assumptions other components can make with respect toward one another, such as provided services, performance characteristics, fault handling, shared resource usage, and so on. The Application Program Interface (API) is the primary mechanism to allow conformity of components. While the slot can accommodate many different component types, the API is the instance that realizes the functionality of a component, and provides cohesion with the overall software architecture. The intent of this definition is that a software architecture must abstract some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and risk reduction.

First, architecture defines types of components. The architecture embodies information about how components must interact with each other to achieve specified system-wide goals. This means that architecture specifically omits content information about components that does not pertain to their interaction.

Second, the definition makes clear that systems can comprise more than one structure and that no one structure holds the irrefutable claim to being *the* architecture. By intention, the definition does not specify what architectural components and relationships are. Is a software component an object? A process? A library? A database? A commercial product? It can be any of these things and more.

Third, the definition implies that every software system, assuming a valid, sustainable, and supportable design, has an architecture, which enables characterization and realization across a family of domain instances, because every system can be shown to be composed of components and relations among them.

Fourth, the realized instance behavior of each component is part of the architecture, insofar as that behavior can be observed or discerned from the point of view of another component. The realized instance of behavior is one aspect of why components can be reused. For example, an infrared tracker component interface instance may differ considerably when implemented in two different missiles, both in terms of hardware interface and periodicity. This polymorphic nature of the API to be utilized within missiles constitutes one method of hardware instance independence, and component level reusability, without necessitating source code modification. This behavior is what allows components to interact with each other, which is clearly part of the architecture.

Finally, placing it all in perspective. The design languages of the software engineer, of which the Unified Modeling Language (UML) is the current mainstream standard, allows control and grouping of 1 . . n objects. Objects in turn control and group anywhere from 1 . . n functions or procedures. Functions or procedures in turn control, and group from 1 . . n lines of source code broken into logical units which have meaning. In each case an abstraction of the physical process is utilized to represent an ever-increasing picture of the overall software system in a comprehensible fashion.

B. WHAT IS MIDDLEWARE?

The term middleware is sometimes used to describe programming that mediates between application and system software or between two different kinds of application software (for example, sending a remote work request from an application in a computer that has one kind of operating system to an application in a computer with a different operating system).

- Reflective Middleware is a term that describes the application of reflection to the engineering of middleware systems in order to achieve openness, configurability and reconfigurability.
- Reflection is the capability of a system to reason about and act upon itself. A reflective system contains a representation of its own behavior, amenable to examination and change, and which is causally connected to the behavior it describes. "Causally-connected" means that changes made to the system's self-representation are immediately reflected in its actual state and behavior, and vice-versa.
- Middleware is a term that refers to a set of services that reside between the application and the operating system, aim to facilitate the development of distributed applications, and function independently of the hardware and operating system.

Systems designed in an event-based architectural style are particularly well suited for distributed environments without central control, to construct component oriented systems, and to support applications that must monitor or react to changes in the environment, information of interest or process status.

The architect of tomorrow's distributed systems will face large-scale distributions both in terms of number of users and connected systems. These systems may include embedded devices, smartcards, mobile phones, handhelds, workstations, mainframes, etc. There is no central administration for the connected parties or for the network itself. Moreover, applications must be deployed spanning organizational and system boundaries, while not being restricted by administrative domains.

Applications must be flexible and extensible in order to fulfill the ever-changing requirements. The overall design must adhere to the paradigm of components everywhere: applications being built today are likely to function as part of bigger integrated systems, tomorrow. With respect to security all parties involved must be assured that consistent policies will be enforced throughout the system. As a trade-off, security aspects influence overall system design and restrict flexibility.

The familiar one-to-one request/reply interaction pattern that is commonly used in client/server systems is inadequate for systems that must react to events representing changes in the environment, information of interest, or process status.

Event-driven task managers best construct applications such as process support systems and workflow management systems using event middleware, realizing the control flow and inter-task dependencies.

In systems like weather alerts and forecasting, stock tracking, news channels, logistics and inventory management, information must be disseminated from many publishers to an even larger number of subscribers, while subscribers must be given the ability to select and extract the data of interest out of a dynamically changing information offer.

Systems supporting mobility have to react to frequent changes in the environment due to movement of mobile devices and users. Moreover, the infrastructure offered by the environment must detect the appearance and disappearance of devices and services adding and removing functionality to the environment.

C. WIDE AREA SYSTEMS

1. Why is scalability such a hard problem?

Perhaps the single-most important problem in developing wide-area distributed systems is handling scalability. Informally, scalability means that we can add users and resources to a system without a noticeable loss in performance or increase of administrative complexity.

Scalability is difficult to quantify, but it is possible to formulate scalability as a precise requirement for distributed systems. In essence one should first decide on what an acceptable performance loss is when users or applications are added, and what it may take to compensate for that loss by adding resources.

Such things may be fine from a formal software engineering point of view, but don't really help when you need to build a scalable wide-area system. Making a system scalable requires that you apply techniques such as caching, replication, and distribution. At the same time, exploiting locality is essential: you don't want your traffic to cross the world when what you're looking for is right next to you.

Given the high degree of system interconnectivity there exists a global synchronization problem. Solving such problems in a general way is inherently difficult and no scalable solutions exist. In other words, applying some scaling techniques, e.g.

replication leads to situations that have only non-scalable solutions (global synchronization).

Of course all is not doom and gloom. For example, in many Web applications, we can safely assume that updates come from a single source and need only to be eventually propagated to replicas. You can have as many copies as you like, or better, as you need to achieve performance. Given also that update propagation does not have to be 100% reliable in many contexts, scalable unreliable multicasting techniques can be used.

Coupling all of the aforementioned with the specialized domain of hard real time embedded systems only amplifies problems.

Likewise, it is also possible to build highly scalable naming services, as demonstrated by Domain Naming Service. DNS combines distribution and caching to ensure locality of operations, and indeed, name lookups are often very fast. User-perceived performance drops when a full name lookup needs to be done in the case of a cache miss.

In effect, scalability sometimes turns out to be less of a problem when considering the application at hand. At the same time, building a general-purpose wide-area system that is inherently scalable may well be unrealistic, given current technology.

To conceive and build universal answers (libraries, architectures, etc.) which attempt to solve all problems for all domains is most likely not the correct answer. Rather ideas must be tailored to fit, and answer specific domain problems.

2. Scalability requires flexibility

One of the main problems is that scaling techniques, such as replication and caching, introduce a global synchronization problem. Having multiple copies simply means that updates have to be propagated to all copies and in the same order to ensure consistency. Consistency is a problem.

If caching and replication are used to scale a system, we need to avoid global synchronization on each update. The solution here is to adopt weaker consistency requirements, if possible. Unfortunately, there is no single rule that applies equally well to all applications.

In order to apply scaling techniques, we need the ability to apply techniques in different ways for different applications. This requirement is putting a hard demand on many Internet applications, notably the Web. In particular, the current Web really does not provide hooks for document-specific, tailor-made solutions. For example, if you want to change your local caching strategy, then this can be done only for *all* cached documents. It's generally not possible to select a few of them based on, for example, source address.

Their is a need for flexibility when it comes to building scalable solutions. What must be done is separate content and functionality from the way we distribute them, and be able to dynamically adapt distribution and replication strategies on a per-case basis.

3. In Summary – Middleware

The role of middleware is to ease the task of designing, programming and managing distributed applications by providing a simple, consistent and integrated distributed programming environment. Essentially, middleware is a distributed software layer, or ‘platform’ which abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages [23].

Different middleware platforms support different programming models. Perhaps the most popular model is object-based middleware in which applications are structured into groupings of (potentially distributed) components composed of objects that interact via location transparent method invocation. Prime examples of this type of middleware are the OMG's CORBA and Microsoft's Distributed COM. Both of these platforms offer an interface definition language (IDL) which is used to abstract over the fact that objects can be implemented in any suitable programming language, an object request broker which is responsible for transparently directing method invocations to the appropriate target object, and a set of services (e.g. naming, time, transactions, replication etc.) which further enhance the distributed programming environment.

Not all middleware is object based, however. Two other popular paradigms are event based middleware and message oriented middleware, both of which mainly employ ‘single shot’ communications rather than the request-reply style communication found in object based middleware. Event based middleware is particularly suited to the construction of non-centralized distributed applications that must monitor and react to changes in their environment. Examples are process control, Internet news channels and stock tracking. It is claimed that event based middleware has potentially better scaling properties for such applications than object based middleware [23]. Message oriented middleware, on the other hand, is biased toward applications in which messages need to be persistently stored and queued. Workflow and messaging applications are good examples.

IV. METHODS AND PRACTICES

A. INCREASED SOPHISTICATION OF DESIGN AND IMPLEMENTATION PRACTICES

There is a saying that history makes a good teacher. Before deciding on a design approach, it would be beneficial to re-examine some key strategies, and concepts related to software development. This chapter seeks to examine risk, complexity management, and development practices leading to a quantification of the fundamental building blocks chosen to implement the software architecture described in this dissertation. The specific contribution made is the description of these same components leading to an embedded hard real time software architecture for missile systems. Previously both RT DII COE and the WSTAWG RT COE have been examined, and rejected as valid approaches. Keeping in mind the domain requirements of missile systems, an analogous approach is derived suitable for the missile domain. The fundamental difference in this approach is that the necessary functionality required for missiles be presented, without the unnecessary additional functionality, and complexity found in RT DII COE, and the tightly coupled hardware requirements of WSTAWG COE.

1. Iterative

Iterative is an adjective that means repetitious.

1) A description of a situation in which a sequence of instructions can be executed multiple times. One pass through the sequence is called iteration. If the sequence of instructions is executed repeatedly, it is called a loop, and we say that the computer iterates through the loop.

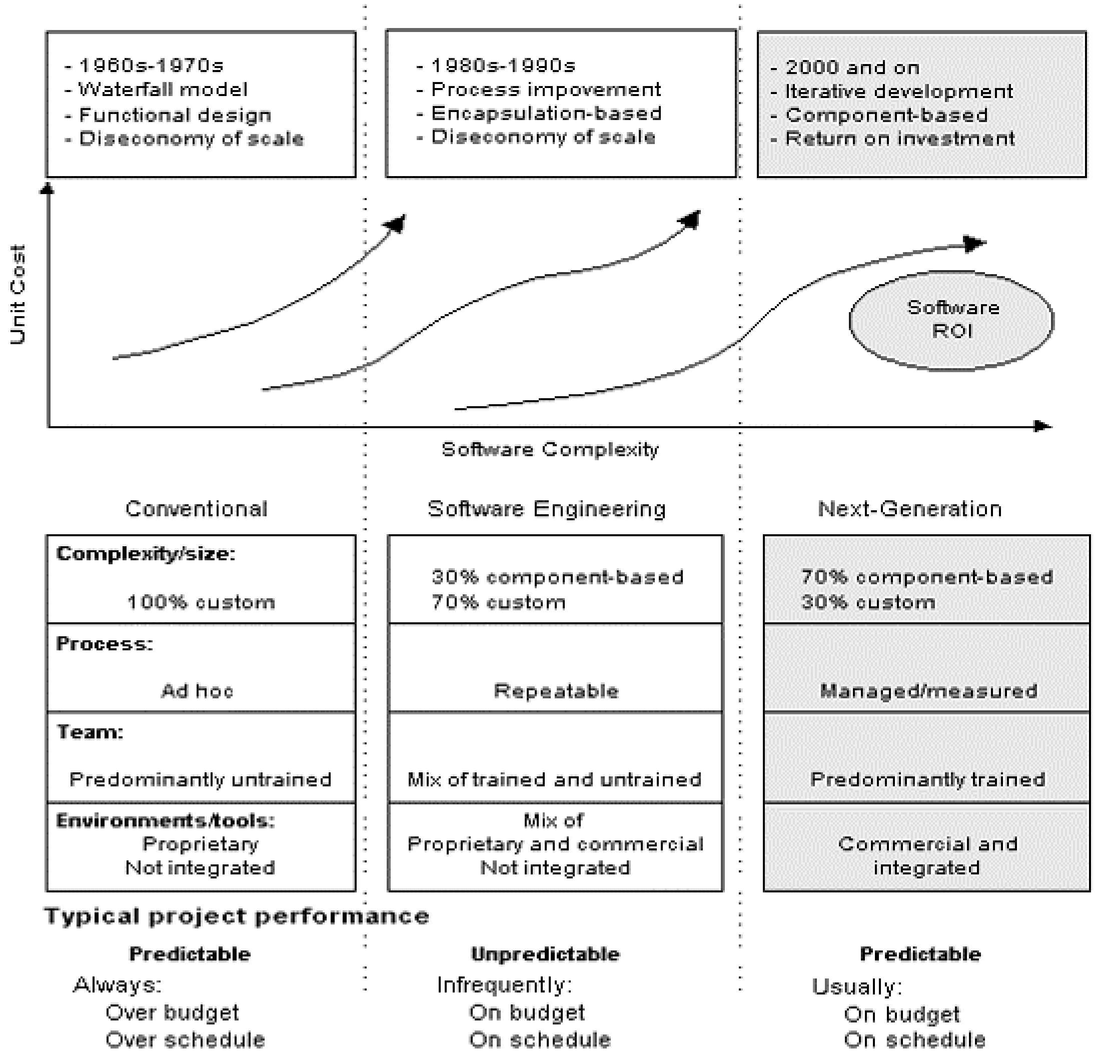
2) In software development, iterative is used to describe a heuristic planning and development process where an application is developed in small sections called iterations. Each iteration is reviewed and critiqued by the software team and potential end-users. Insights gained from the critique of iteration are used to determine the next step in development. Data models or sequence diagrams, which are often used to map out iterations, keep track of what has been tried, approved, or discarded, and eventually serve as a kind of blueprint for the final product.

The challenge in iterative development is to make sure all the iterations are compatible. As each new iteration is approved, developers may employ a technique known as backwards engineering, which is a systematic review and check procedure to make sure each new iteration is compatible with previous ones. The advantage of using iterative development is that the end-user is involved in the development process. Instead of waiting until the application is a final product, when it may not be possible to make changes easily, problems are identified and solved at each stage of development. Iterative development is sometimes called circular or evolutionary development.

2. Trends in the Software Science

Software engineering is dominated by intellectual activities focused on solving problems with immense complexity and numerous unknowns in competing perspectives. In the early software approaches of the 1960s and 1970s, craftsmanship was the key factor for success; each project used a custom process and custom tools. In the 1980s and 1990s, the software industry matured and transitioned into more of an engineering discipline. However, most software projects in this era were still primarily research-intensive, dominated by human creativity and diseconomies of scale. Today, the next generation of software processes is driving toward a more production-intensive approach, dominated by automation and economies of scale. Further characterization these three generations of software development as follows:

1. **1960s and 1970s: Conventional.** Organizations used virtually all custom tools, custom processes, and custom components built in primitive languages. Project performance was highly predictable: Cost, schedule, and quality objectives were almost never met.
2. **1980s and 1990s: Software engineering.** Organizations used more repeatable processes, off-the-shelf tools, and about 70 percent of their components were built in higher level languages. About 30 percent of these components were available as commercial products, including the operating system, database management system, networking, and graphical user interface. During the 1980s, some organizations began achieving economies of scale, but with the growth in applications complexity (primarily in the move to distributed systems), the existing languages, techniques, and technologies were just not enough.
3. **2000 and later: Next generation.** Modern practice is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70 percent) off-the-shelf components. Typically, only about 30 percent of components need to be custom built once the baseline functionality is established, i.e. a component library.



Conventional	Software Engineering	Next-Generation
Complexity/size: 100% custom	30% component-based 70% custom	70% component-based 30% custom
Process: Ad hoc	Repeatable	Managed/measured
Team: Predominantly untrained	Mix of trained and untrained	Predominantly trained
Environments/tools: Proprietary Not integrated	Mix of Proprietary and commercial Not integrated	Commercial and integrated

Typical project performance

Predictable	**Unpredictable**	**Predictable**
Always: Over budget Over schedule	Infrequently: On budget On schedule	Usually: On budget On schedule

Figure 6. Trends in Software Development [30]

Figure 6 illustrates the economics associated with these three generations of software development. The ordinate of the graph refers to software unit costs (per Source Line of Code (SLOC), per function point, per component -- take your pick) realized by an organization. Equivalent Line of Code (ELOC) is a more contemporary approach to level the playing field by equating difficulty with a standard measure -- line of code [30].

Technologies for achieving reductions in complexity/size, process improvements, improvements in team proficiency, and tool automation are not independent of one another. In each new generation, the key is complementary growth in all technologies. For example, the process advances could not be used successfully without new component technologies and increased tool automation.

B. REDUCING SOFTWARE PRODUCT SIZE OR COMPLEXITY

1. Reducing the Size of Human-Generated Code

Component-based technology is a general term for reducing the size and complexity of human-generated code necessary to achieve a software solution. Commercial components, domain-specific reuse, architectural patterns, and higher-order programming languages are all elements of component-based approaches focused on achieving a given system with fewer lines of human-specified source directives (statements). For example, to achieve a certain application with a fixed number of features, we could use any of the following potential solutions:

- Develop 1,000,000 lines of custom assembly language.
- Develop 400,000 lines of custom C++.
- Develop 100,000 lines of custom C++, integrate 200,000 lines of existing reusable components, and purchase a commercial middleware product.
- Develop 50,000 lines of custom Visual Basic, and purchase and integrate several commercial components on a Win32 platform.
- Develop 5,000 lines of custom Java, develop 10,000 lines of custom HTML, and purchase and integrate several commercial components on a Java platform.

While the temptation to integrate Commercial of the Shelf (COTS) software is great, it is important to understand the tradeoffs which must be weighed to fully understand a decision – either way. Each of these solutions represents a step up in exploiting component based technology and a commensurate reduction in the total amount of human-developed code, which in turn reduces the time and the team size needed for development. Since the difference between large and small projects has a greater than linear impact on the life-cycle cost, the use of the highest-level language and appropriate commercial components has the highest potential cost impact. Furthermore, simpler is generally better. Reducing the size of custom-developed software usually increases understandability, reliability, and the ease of making changes.

2. Managing Complexity Through Visual Modeling

Object-oriented technology and visual modeling achieved rapid acceptance during the 1990s. The fundamental impact of object-oriented technology has been in reducing the overall size and complexity of what needs to be developed through more formalized notations for capturing and visualizing software abstractions. A model is a simplification of reality that provides a complete description of a system. We build models of complex systems because we cannot comprehend any such system in its entirety. Modeling is important because it helps the development team visualize, specify, construct, and communicate the structure and behavior of a system's architecture.

Using a standard modeling notation such as the Unified Modeling Language (UML) [28], different members of the development team can communicate their decisions unambiguously to each other. Using visual modeling tools facilitates the management of these models, letting you hide or expose details as necessary. Visual modeling also helps maintain consistency among a system's artifacts: its requirements, designs, and implementations. In short, visual modeling helps improve a team's ability to manage software complexity.

C. IMPROVING THE DEVELOPMENT PROCESS

In order to achieve success, real-world software projects require an incredibly complex web of sequential and parallel steps. As the scale of the project increases, more overhead steps must be included just to manage the complexity of this web. All project processes consist of productive activities and overhead activities.

- Productive activities result in tangible progress toward the end product. For software efforts, these activities include prototyping, modeling, coding, integration, debugging, and user documentation.
- Overhead activities have an intangible impact on the end product. They include plan preparation, requirements management, documentation, progress monitoring, risk assessment, financial assessment, configuration control, quality assessment, testing, late scrap and rework, management, personnel training, business administration, and other tasks. Although overhead activities include many value-added efforts, in general, when less effort is devoted to these activities, more effort can be expended on productive activities. While many of the overhead activities are unaviodable, the main thrust of process improvement is to improve the results of productive activities and minimize the impact of overhead activities on personnel and schedule. Based on our observations, these are the three most quantifiable approaches for achieving significant process improvements:
    1. Transitioning to an iterative process.
    2. Attacking the significant risks first through a component-based, architecture-first focus.

3. Using key software engineering best practices, from the outset, in requirements management, visual modeling, change management, and assessing quality throughout the life cycle.

1. Using an Iterative Process

The key discriminator in significant process improvement is making the transition from the conventional (waterfall) approach to a modern, iterative approach. The conventional software process was characterized by transitioning through sequential phases, from requirements to design to code to test, achieving 100 percent completion of each artifact at each life-cycle stage. All requirements, artifacts, components, and activities were treated as equals. The goal was to achieve high-fidelity traceability among all artifacts at each stage in the life cycle. In practice, the conventional process result in [57]:

- Protracted integration and late design breakage.
- Late risk resolution.
- Requirements-driven functional decomposition.
- Adversarial stakeholder relationships.
- Lack of effective communications
- Focus on documents and review meetings.

These symptoms almost always led to a significant diseconomy of scale, especially for larger projects involving many developers. By contrast, a modern (iterative) development process framework is characterized by [28]:

1. Continuous round-trip engineering from requirements to test, at evolving levels of abstraction.

2. Achieving high-fidelity understanding of the architecturally significant decisions as early as practical.
3. Evolving the artifacts in breadth and depth based on risk management priorities.
4. Postponing completeness and consistency analyses until later in the life cycle.

A modern process framework attacks the primary sources of the diseconomy of scale inherent in the conventional software process. Figure 7 provides an objective perspective of the difference between the conventional waterfall process and a modern iterative process. It graphs development progress versus time, where progress is defined as percent coded, that is, demonstrable in its target form. However, in the Waterfall process, the target form is based upon firm, fixed requirements, which rarely, if ever, happen in the real world. At that point, the software is compilable and executable. It is not necessarily complete, compliant, nor up to specifications, as the testing, verification, and validation do not occur until late in the process, during integration.

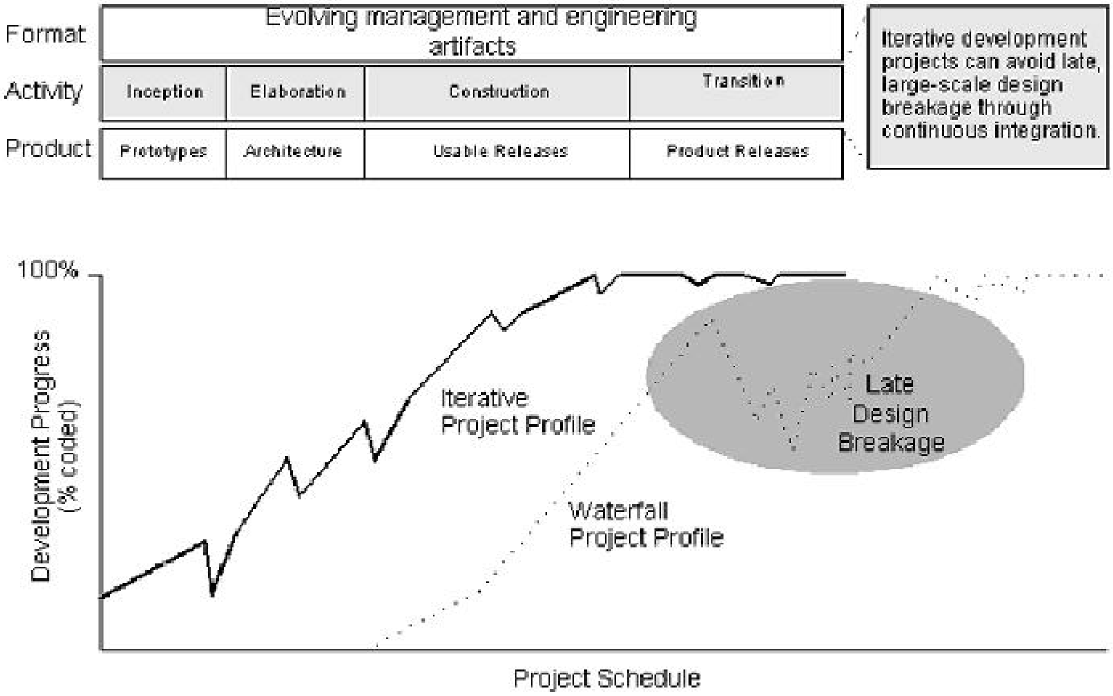


Figure 7. Project Profiles for the Waterfall and Iterative Processes [60]

2. Historical Software Development – The Waterfall Process

Requirements were first captured in complete detail in ad hoc text. Design documents were then fully elaborated in ad hoc notations. Coding and unit testing of individual components followed. Finally, the components were compiled and linked together into a complete system. This integration activity was the first time that significant inconsistencies among components (their interfaces and behavior) could be tangibly recognized. These inconsistencies, some of which were extremely difficult to uncover, resulted from using ambiguous formats for the early life-cycle artifacts. Getting the software to operate reliably enough to test its usefulness took much longer than planned. Budget and schedule pressure drove teams to shoehorn in the quickest fixes; redesign was usually out of the question. Then the testing of system threads, usefulness, requirements compliance, and quality was performed through a series of releases until the software was judged adequate for the user. About 90 percent of the time, the end result was a software system that was late, over budget, fragile, and expensive to maintain. The sad truth is that such systems have propagated themselves throughout a product life cycle, adding untold cost, and schedule impact [60].

A review of numerous conventional projects that followed a waterfall model shows a recurring symptom: Although it was not usually planned this way, the resources expended in the major software development workflows resulted in an excessive allocation of resources (time or effort) to accomplish integration and test. Successfully completed projects consumed 40 percent of their effort in these activities; the percentage was even higher for unsuccessful projects. The overriding reason was that the effort associated with the late scrap and rework of design flaws was collected and implemented during the integration and test phases. Integration is a non-value-added activity, and most integration and test organizations spent 60 percent of their time integrating, that is, getting the software to work by resolving the design flaws and the frequent scrap and rework associated with these resolutions. It is preferable for integration to take little time and little effort so the integration and test team can focus on demonstrating and testing the software, which are value-added efforts.

3. Attacking Significant Risks Early

Using an iterative development process, the software development team produces the architecture first, allowing integration to occur as the "verification" activity of the design phase and allowing design flaws to be detected and resolved earlier in the life cycle. This replaces the big-bang integration at the end of a project with continuous integration throughout the project. Getting the architecturally important things to be well understood and stable before worrying about the complete breadth and depth of the artifacts should result in scrap and rework rates that decrease or remain stable over the project life cycle [34].

The architecture-first approach forces integration into the design phase and demonstrations provide the forcing function for progress. The demonstrations do not eliminate the design breakage; they just make it happen in the design phase where it can be fixed correctly. In an iterative process, the system is "grown" from an immature prototype to a baseline architectural skeleton to increments of useful capabilities to finally complete product releases. The downstream integration nightmare is avoided, and a more robust and maintainable design is produced. Major milestones provide very tangible results. Designs are now guilty until proven innocent: The project does not move forward until the objectives of the demonstration have been achieved. Results of the demonstration and major milestones contribute to an understanding of the tradeoffs among the requirements, design, plans, technology, and other factors. Based on this understanding, changes to stakeholder expectations can still be renegotiated. The early phases of the iterative life cycle (inception and elaboration) focus on confronting and resolving the risks before making the big resource commitments required in later phases.

Managers of conventional projects tend to do the easy stuff first, thereby demonstrating early progress. A modern process, as shown in Figure 8, needs to attack the architecturally significant stuff first, the important 20 percent of the requirements: use cases, components, and risks.

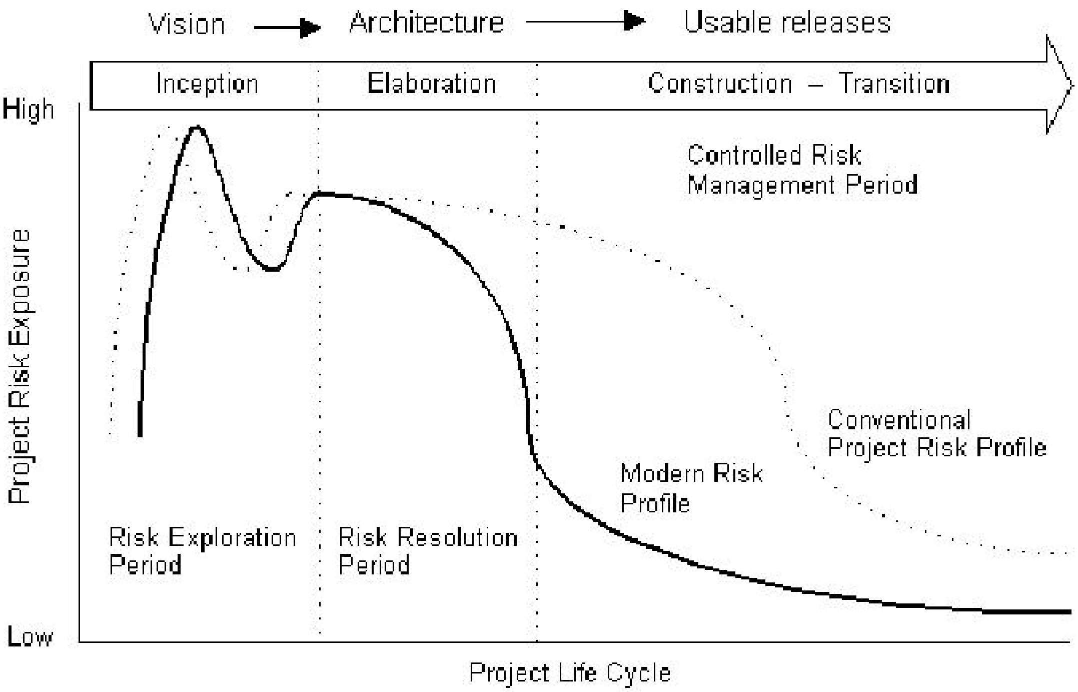


Figure 8. Architecture First, Then Production [60]

The "80/20" lessons learned during the past 30 years of software management experience provide a useful perspective for identifying some of the key features of an iterative development philosophy [28][34][60]:

- 80% of the engineering is consumed by 20% of the requirements. Do not strive prematurely for high fidelity and full traceability of the complete requirements set. Instead, strive to understand the driving requirements completely before committing resources to full-scale development.
- 80% of the software cost is consumed by 20% of the components. Elaborate the cost-critical components first so that planning and control of cost drivers are well understood early in the life cycle.

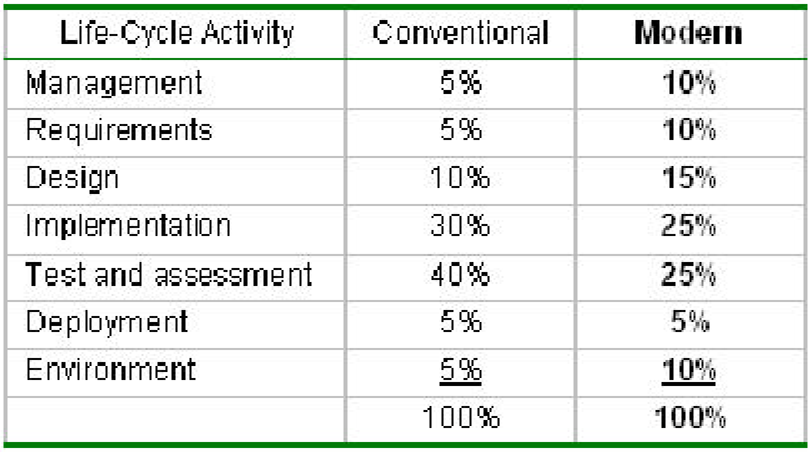
- 80% of the errors are caused by 20% of the components. Elaborate the reliability-critical components first so that assessment activities have enough time to achieve the necessary level of maturity.
- 80% of software scrap and rework is caused by 20% of the changes. Elaborate the change-critical components first so that broad-impact changes occur when the project is nimble.
- 80% of the resource consumption (execution time, disk space, and memory) is consumed by 20% of the components. Elaborate the performance-critical components first so that engineering tradeoffs with reliability, changeability, and cost effectiveness can be resolved as early in the life cycle as possible.
- 80% of the progress is made by 20% of the people. Make sure the initial team that plans the project and designs the architecture is of the highest quality. An adequate plan and architecture can then succeed with an average construction team. An inadequate plan or inadequate architecture will probably not succeed, even with an expert construction team.

4. Using Software Best Practices

Best practices are a set of commercially proven approaches to software development that, when used together, strike at the root causes of software development problems. The Rational Unified Process (RUP) integrates six industry best practices into one process framework [34]:

1. Develop iteratively
2. Manage requirements
3. Use component architectures
4. Model visually
5. Continuously verify quality
6. Manage change

The techniques and technologies inherent in these best practices are discussed in detail in the RUP. One way to view the impact of these best practices on the economics of software projects is through the differences in resource expenditure profiles between conventional projects and modern iterative projects. Conventional principles drove software development activities to overextend during implementation and integration activities. A healthy iterative process, an architecture-first focus, and incorporation of software best practices should result in less total scrap and rework through relatively more emphasis on the high-value activities of management planning, requirements analysis, and design. This results in a more balanced expenditure of resources across the core workflow of a modern process [28].



Life-Cycle Activity	Conventional	**Modern**
Management	5%	**10%**
Requirements	5%	**10%**
Design	10%	**15%**
Implementation	30%	**25%**
Test and assessment	40%	**25%**
Deployment	5%	**5%**
Environment	5%	**10%**
	100%	**100%**

Table 1. Resource Expenditures

One critical lesson learned from successful iterative development projects is that they start out with a planning profile different from the standard profile of a conventional project. If you plan modern iterative projects with the waterfall planning profile, the chance of success is significantly diminished. By planning a modern project with a more appropriate resource profile derived from successful iterative projects, there is much more flexibility in optimizing the project performance for improvements in productivity, quality, or cycle time, whichever is the business driver. Typical goals are to achieve a 2X, 3X, or 10X increase in productivity, quality, time to market, or some combination of all three, where X corresponds to how well the organization does now. The funny thing is that most of these organizations have only a coarse grasp on what X is, in objective terms. Table 1 characterizes the impact on project expenditure profiles associated with making about a 3X reduction in scrap and rework. This improvement is the primary goal of transitioning from the conventional waterfall software development process to a modern iterative software development process. Standardizing on a common process is a courageous undertaking for a software organization, and there is a wide spectrum of implementations. I have seen organizations attempt to do less (too little standardization, or none) and more (too much standardization) with little success in improving software return on investment. Process standardization requires a very balanced approach.

D. THE COMMUNICATIONS PARADIGM

Best practices, Waterfall, and the iterative development methodologies have been examined. Keeping in mind what has been learned over the course of software development history, it is now appropriate to begin describing the necessary requirements – services of the embedded hard real time software architecture. Upon completion of these descriptions, it will then be possible to design the physical characteristics of the software architecture itself.

Interprocess / processor communications within software systems typically fall into three paradigms. The underlying support mechanisms such as Message Passing Interface (MPI), RapidIO, Infiniband, and others represent the implementation [30].

Distributed Active Systems (DAS) support building applications that must monitor and react to changes in the environment, information of interest or process status. The publish/subscribe interaction paradigm is at the heart of DAS. Notification services provided may range from simple messaging to content based filtering and event composition and encompass integration with transaction mechanisms and support transactional coupling.

Ubiquitous and Mobile Systems (UMS) cover applications characterized by the heterogeneity of systems and devices, as well as the (spontaneous) patterns of interconnection. Moreover, contrary to traditional distributed problem solving, there is not only one global problem to be solved but also, due to the user-centric nature, a number of local problems. Relevant issues are naming, service trading, context- and location awareness. Due to the unpredictably of interaction schemes and the preferred use of asynchronous communication patterns, system design based on the notion of events seems to be superior to classical client/server interaction schemes.

Message Oriented Middleware (MOM) is a specific class of middleware that supports the exchange of general-purpose messages in a distributed application environment. There is a wide spectrum of MOM encompassing publish/subscribe based systems and message queuing systems. MOM supports data exchange and request/reply style interaction by publishing messages and/or message queuing in a synchronous and asynchronous (connectionless) manner. The MOM system ensures message delivery by using reliable (and persistent) queues or reliable multicast and provides directory, security, and administrative services.

Systems designed in an event-based architectural style are particularly well suited for distributed environments without central control, to construct component oriented systems, and to support applications that must monitor or react to changes in the environment, information of interest or process status.

The architect of tomorrow's distributed systems will face large-scale distributions both in terms of number of users and connected systems. These systems may include embedded devices, smartcards, mobile phones, handhelds, workstations, mainframes, etc. There is no central administration for the connected parties or for the network itself. Moreover, applications must be deployed spanning organizational and system boundaries, while not being restricted by administrative domains.

1. The Shared Memory Model of Interprocess Communication

Information exchange between processes (interprocess) and physical processors (interprocessor) communication is critical when operating in a distributed environment. In the case of missiles, the physical hardware implementation is not guaranteed between iterations of one product enhancement / upgrade to another. It is therefor very important that the methods chosen for interprocessor / interprocess communication not have a physical dependency of any particular hardware implementation. The shared memory architecture to be described is a combination of two distinct methods for communications. The first, shared memory has the advantage of being abstracted from the physical details, with actual memory addressing handled by proxy abstraction. The second method, which is coupled to the first, via a middleware instance is direct memory access (DMA). DMA channelization has the advantage of being extremely fast, but is tied to the specific hardware being utilized. DMA channelization is realized at the OS BSP level, where the details of implementation occur. The higher level middleware services provide non-specific memory addressing mechanisms, utilizing the DMA addressing required for the specific hardware instance. Benchmark tests confirm an increase of over 13x when shared memory is coupled to DMA channelization. (Shared memory only, via a PCI bridge @66Mhz – 32 bit words, has a throughput of 6 Mbps, while adding DMA channelization yields results in the 100Mbps range). The unique contribution here is the coupling of abstract methods with the advantages of specific hardware implementation to obtain otherwise improbable results, while maintaining hardware independence.

All interprocess and device communication is provided in the caching model by implementing it as an extension of the virtual memory system using memory-based messaging [7]. With memory-based messaging, threads communicate through the memory system by mapping a shared region of physical memory into the sender and receiver address spaces, as illustrated in Figure 9.

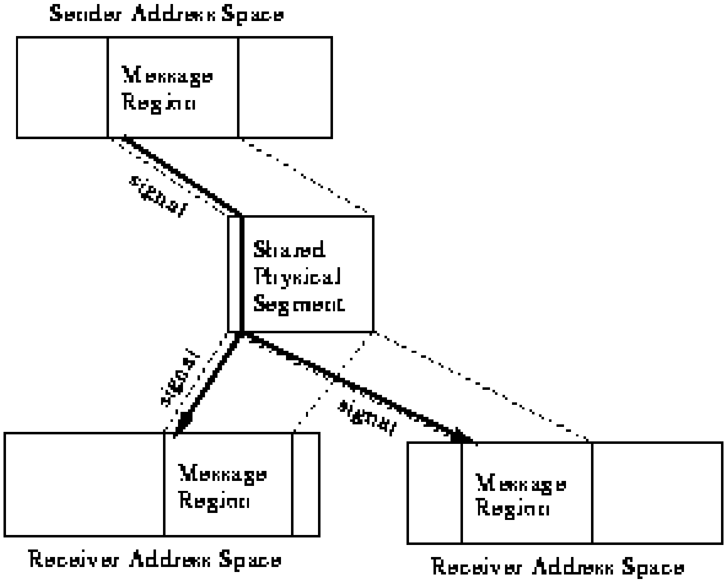


Figure 3: Memory-based Messaging

Figure 9. Memory based Messaging

The sending thread writes a message into this region and then delivers the address of the new message to the receiving threads as an address-valued signal. That is, the virtual address corresponding to the location of the new message is passed to the receiving threads' signal function, translated from the virtual address of the sending thread. On receiving the address-valued signal, the receiving thread reads the message at the designated location in the virtual memory region. While the thread is running in its signal function, additional signals are queued within the cache kernel.

To support memory-based messaging, the messages are extended to optionally specify a signal thread and also to specify that the shared memory address is in message mode. An application kernel interested in receiving signals for a given address specifies a signal thread in the mapping for the address space. The signaling uses the same mapping data structures as the rest of the virtual memory system. The performance-critical data transfer aspect of interprocess communication is performed directly through the memory system.

Communication performance is limited primarily by the raw performance of the memory system, not the software overhead of copying, queuing and delivering messages.

Memory-based messaging is used for accessing devices controlled by the cache kernel. For example, an Ethernet device is a possible solution to provide the necessary memory-mapped transmission and reception memory regions. The client thread sends a signal to the Ethernet driver in the cache kernel to transmit a packet with the signal address indicating the packet buffer to transmit. On reception, a signal is generated to the receiving thread with the signal address indicating the buffer holding the new packet. This thread demultiplexes the data to the appropriate input stream, similar to conventional network protocol implementations.

Devices that fit into the memory-based messaging model directly require minimal driver code complexity of the cache kernel. They also provide optimal performance. In particular, the driver only needs to support memory mapping the special device address space corresponding to the network interface. Data transfer and signaling is then handled using the general cache kernel memory-based messaging mechanism. The clock is also designed to fit this memory-based messaging model. In contrast, the Ethernet device requires a non-trivial cache kernel driver to implement the memory-based messaging interface because the Ethernet chip itself provides a conventional DMA interface.

An object-oriented Remote Procedure Call (RPC) facility implemented on top of the memory-based messaging as a user-space communication library allows applications and services to use a conventional procedural communication interface to services. For instance, object writeback from the cache kernel to the owning application kernel uses a writeback channel implemented using this facility. This RPC facility is also used for high-performance communication between distributed application kernels. Memory-based messaging supports direct marshaling and demarshaling to and from communication channels with minimal copying and no protection boundary crossing in software. The implementation in user space allows application kernels control over communication resource management and exception handling.

2. Necessary services

Middleware is connectivity software that consists of a set of enabling services that allow multiple processes running on one or more machines to interact across a network. Middleware is essential to migrating mainframe applications to client/server applications and providing communication across heterogeneous platforms. This technology has evolved during the 1990s to provide interoperability in support of the move to client/server architectures. The most widely-publicized middleware initiatives are the

Open Software Foundation's Distributed Computing Environment (DCE), Object Management Group's Common Object Request Broker Architecture (CORBA), and Microsoft's COM/DCOM

As outlined in Figure 10, middleware services are sets of distributed software that exist between the application and the operating system and network services on a system node in the network.

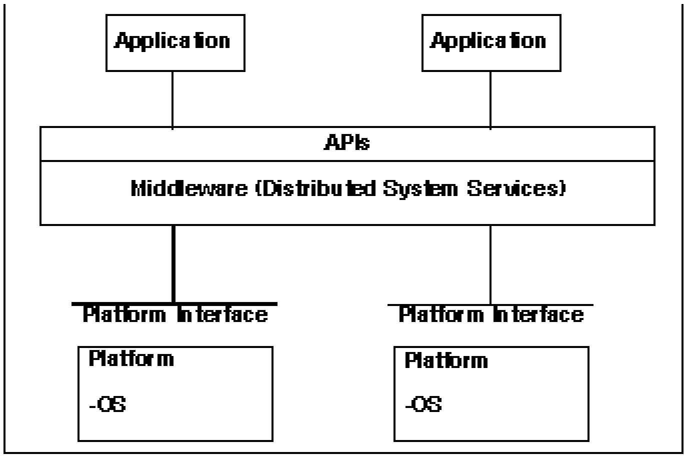


Figure 10. Use of Middleware

Middleware services provide a more functional set of Application Programming Interfaces (API) than the operating system and network services to allow an application to

- locate transparently across the network, providing interaction with another application or service
- be independent from network services
- be reliable and available
- scale up in capacity without losing function

Middleware can take on the following different forms:

- Transaction processing (TP) monitors, which provide tools and an environment for developing and deploying, distributed applications.
- Remote Procedure Call (RPCs), which enable the logic of an application to be distributed across the network. Program logic on remote systems can be executed as simply as calling a local routine.
- Message-Oriented Middleware (MOM), which provides program-to-program data exchange, enabling the creation of distributed applications. MOM is analogous to email in the sense it is asynchronous and requires the recipients of messages to interpret their meaning and to take appropriate action.
- Object Request Brokers (ORBs) , which enable the objects that comprise an application to be distributed and shared across heterogeneous networks.

E. POLYMORPHIC APPLICATION PROGRAM INTERFACE

Interface specifications are critical for component-based software architecture to function correctly. While it is desirable to have an exact specification for each interface, future growth, and expansion must be accommodated. To this end the concept of the polymorphic application program interface is presented. While all the normal capabilities of inheritance, abstraction, and over rides are available within the context of an API, the polymorphic capabilities of object oriented programming are also available for inclusion.

1. API - application programming interface

An API is a series of functions that programs can use to make the operating system do their dirty work. Using Windows APIs, for example, a program can open windows, files, and message boxes--as well as perform more complicated tasks--by passing a single instruction. Windows has several classes of APIs that deal with telephony, messaging, and other issues.

2. MAPI - Messaging Application Programming Interface

Microsoft and other companies developed MAPI (pronounced "mappy") to enable Windows applications to access a variety of messaging systems, from Microsoft Mail to Novell's MHS. But MAPI works on more everyday level, so-called mail-aware applications can exchange both mail and data with others on a network.

3. Polymorphism

Polymorphism is the ability to appear in many forms. In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes. For example, given a base class shape, polymorphism enables the programmer to define different circumference methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the circumference method to it will return the correct results. Polymorphism is considered to be a requirement of any true object-oriented programming language (OOPL) [28].

The type of polymorphism described above is sometimes called parametric polymorphism to distinguish it from another type of polymorphism called overloading.

F. FUNDAMENTALS OF MIDDLEWARE – BUILDING BLOCKS FOR ABSTRACTION

There are a small number of different kinds of middleware that have been developed. These vary in terms of the programming abstractions they provide and the kinds of heterogeneity they provide beyond network and hardware.

1. Message-Oriented Middleware

Message-Oriented Middleware (MOM) provides the abstraction of a message queue that can be accessed across a network. It is a generalization of the well-known operating system construct: the mailbox. It is very flexible in how it can be configured with the topology of programs that deposit and withdraw messages from a given queue. Many MOM products offer queues with persistence, replication, or real-time performance.

2. Distributed Object Middleware

Distributed object middleware provides an abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques encapsulation, inheritance, and polymorphism available to the distributed application developer.

The Common Object Request Broker Architecture is a standard for distributed object computing. It is part of the Object Management Architecture (OMA), developed by the Object Management Group (OMG), and is the broadest distributed object middleware available in terms of scope. It encompasses not only CORBA's distributed object abstraction but also other elements of the OMA which address general purpose and vertical market components helpful for distributed application developers. CORBA offers heterogeneity across programming language and vendor implementations. CORBA (and the OMA) is considered by most experts to be the most advanced kind of middleware commercially available and the most faithful to classical object oriented programming principles. Its standards are publicly available and well defined.

DCOM is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) and Component Object Model (COM). DCOM's distributed object abstraction is augmented by other Microsoft technologies, including Microsoft Transaction Server and Active Directory. DCOM provides heterogeneity across language but not across operating system or tool vendor. COM+ is the next-generation DCOM that greatly simplifies the programming of DCOM. SOAP is a distributed object framework from Microsoft that is based on XML and HyperText Transfer Protocols (HTTP). Its specification is public, and it provides heterogeneity across both language and vendor. Microsoft's distributed object framework .NET also has heterogeneity across language and vendor among its stated goals.

Java has a facility called Remote Method Invocation (RMI) that is similar to the distributed object abstraction of CORBA and DCOM. RMI provides heterogeneity across operating system and Java vendor, but not across language. However, supporting only Java allows closer integration with some of its features, which can ease programming and provide greater functionality.

3. Marketplace Convergence of the Concepts

The categories of middleware above are blurred in the marketplace in a number of ways. Starting in the late 1990s, many products began to offer APIs for multiple abstractions, for example distributed objects and message queues, and managed in part by a TPM. TPMs in turn often use RPC or MOM as an underlying transport while adding management and control facilities. Relational database vendors have been breaking the relational model and the strict separation of data and code by many extensions, including RPC-like stored procedures. To complicate matters further, Java is being used to program these stored procedures. Additionally, some MOM products offer transactions over multiple operations on a message queue. Finally, distributed object systems typically offer event services or channels that are similar to MOM in term of architecture, namely topology and data flow.

4. Middleware and Legacy Systems

Middleware is sometimes called a “glue” technology because it is often used to integrate components. It is essential for migrating mainframe applications that were never designed to interoperate or be networked to service remote requests. Middleware is also very useful for wrapping network devices such as routers and mobile base stations to offer network integrators and maintainers a control API that provides interoperability at the highest level. Distributed object middleware is particularly well suited for legacy integration, due to its generality. In short, it provides a very high lowest common denominator of interoperability. CORBA, in particular, is typically used for this because it supports the most kinds of heterogeneity and thus allows the legacy components to be used as widely as possible.

5. Programming with Middleware

Programmers do not have to learn a new programming language to program with middleware. Rather, they use an existing one they are familiar with, such as C++ or Java. There are three main ways in which middleware can be programmed with existing languages. The first is where the middleware system provides a library of functions to be called to utilize the middleware; distributed database systems and inter process / processor communications do this. The second is through an external interface definition language. In this approach, the IDL file describes the interface to the remote component, and a mapping from the IDL to the programming language is used for the programmer to code to. The third way is for the language and runtime system to support distribution natively; for example, Java's Remote Method Invocation (RMI).

6. Middleware and Layering

There may be multiple layers of middleware present in a given system configuration. For example, lower-level middleware such as a virtually synchronous atomic broadcast service can be used directly by application programmers. However, sometimes it is used as a building block by higher-level middleware such as CORBA or Message-Oriented Middleware to provide fault tolerance or load balancing or both. Note that most of the implementation of a middleware system is at the "Application" Layer 7 in the OSI network reference architecture [65], though parts of it is also at the "Presentation" Layer 6. Thus, the middleware is an "application" to the network protocols, which are in the operating system. The "application" from the middleware's perspective is above it.

Layer	Name	Functional Description
7	Application Layer	Provides network services to the end-users. Mail, ftp, telnet, DNS, NIS, NFS are examples of network applications.
6	Presentation Layer	External Data Representation (XDR) sits at the presentation level. It converts local representation of data to its canonical form and vice versa. The canonical uses a standard byte ordering and structure packing convention, independent of the host.
5	Session Layer	The session protocol defines the format of the data sent over the connections. The NFS uses the Remote Procedure Call (RPC) for its session protocol. RPC may be built on either TCP or UDP. Login sessions uses TCP whereas NFS and broadcast use UDP.
4	Transport Layer	Transport layer subdivides user-buffer into network-buffer sized datagrams and enforces desired transmission control. Two transport protocols, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), sits at the transport layer. Reliability and speed are the primary difference between these two protocols. TCP establishes connections between two hosts on the network through 'sockets' which are determined by the IP address and port number. TCP keeps track of the packet delivery order and the packets that must be resent. Maintaining this information for each connection makes TCP a stateful protocol. UDP on the other hand provides a low overhead transmission service, but with less error checking. NFS is built on top of UDP because of its speed and statelessness. Statelessness simplifies the crash recovery.
3	Network Layer	NFS uses Internetwork Protocol (IP) as its network layer interface. IP is responsible for routing, directing datagrams from one network to another. The network layer may have to break large datagrams, larger than Maximum Transmission Unit (MTU), into smaller packets and host receiving the packet will have to reassemble the fragmented datagram. The Internetwork Protocol identifies each host with a 32-bit IP address. IP addresses are written as four dot-separated decimal numbers between 0

and 255, e.g., 129.79.16.40. The leading 1-3 bytes of the IP identify the network and the remaining bytes identifies the host on that network. The network portion of the IP is assigned by InterNIC Registration Services, under the contract to the National Science Foundation, and the local network administrators assign the host portion of the IP. For large sites, usually subnetted like ours, the first two bytes represents the network portion of the IP, and the third and fourth bytes identify the subnet and host respectively.

Even though IP packets are addressed using IP addresses, hardware addresses must be used to actually transport data from one host to another. The Address Resolution Protocol (ARP) is used to map the IP address to it hardware address.

2 Data Link Layer

Data Link layer defines the format of data on the network. A network data frame, aka packet, includes checksum, source and destination address, and data. The largest packet that can be sent through a data link layer defines the Maximum Transmission Unit (MTU). The data link layer handles the physical and logical connections to the packet's destination, using a network interface. A host connected to an Ethernet would have an Ethernet interface to handle connections to the outside world, and a loopback interface to send packets to itself.

Ethernet addresses a host using a unique, 48-bit address called its Ethernet address or Media Access Control (MAC) address. MAC addresses are usually represented as six colon-separated pairs of hex digits, e.g., 8:0:20:11:ac:85. This number is unique and is associated with a particular Ethernet device. Hosts with multiple network interfaces should use the same MAC address on each. The data link layer's protocol-specific header specifies the MAC address of the packet's source and destination. When a packet is sent to all hosts (broadcast), a special MAC address (ff:ff:ff:ff:ff:ff) is used.

.

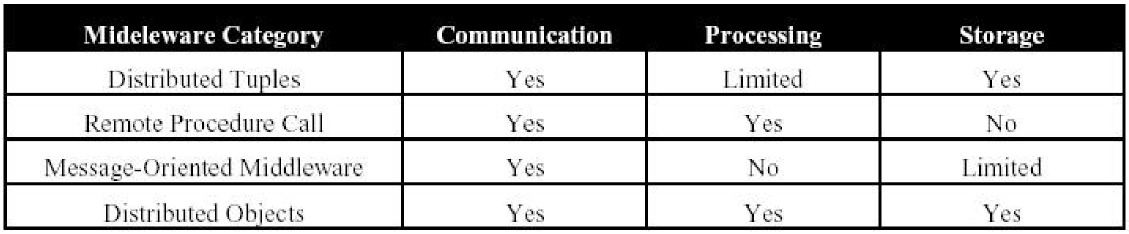
1 Physical Layer

Physical layer defines the cable or physical medium itself, e.g., thinnet, thicknet, unshielded twisted pairs (UTP). All media are functionally equivalent. The main difference is in convenience and cost of installation and maintenance. Converters from one media to another operate at this level.

Table 2 OSI / ISO Network Reference Architecture

7. Middleware and Resource Management

The abstractions offered by various middleware frameworks can be used to provide resource management in a distributed system at a higher level than is otherwise possible. This is because these abstractions can be designed to be rich enough to subsume the three kinds of low-level physical resources an operating system manages: communications, processing, and storage (memory and disks). Middleware's abstractions also are from an end-to-end perspective, not just that of a single host, which allows for a more global and complete view to a resource management system. All middleware-programming abstractions by definition subsume communications resources, but others vary in how well they incorporate processing and storage. Table 3 depicts how well each category of middleware encapsulates and integrates these resources. Distributed tuples offer only a limited form of processing to the client. RPC does not integrate storage, while MOM does not include processing.



Mideleware Category	Communication	Processing	Storage
Distributed Tuples	Yes	Limited	Yes
Remote Procedure Call	Yes	Yes	No
Message-Oriented Middleware	Yes	No	Limited
Distributed Objects	Yes	Yes	Yes

Table 3: Middleware Encapsulation and Integration of Low-Level Resources

Distributed objects, however, not only encapsulate but cleanly integrate all three kinds of resource into a coherent package. This completeness helps distributed resource management but also makes it easier to provide different kinds of distributed transparencies such as mobility transparency [26][30]

8. Middleware and Quality of Service Management

Distributed systems are inherently very dynamic, which can make them difficult to program. Resource management is helpful, but is generally not enough for most distributed applications. Starting in the late 1990s, distributed systems research has begun to focus on providing comprehensive quality of service (QoS), an organizing concept referring to the behavioral properties of an object or system, to help manage the dynamic nature of distributed systems. The goal of this research is to capture the application's high-level QoS requirements and then translate them down to low-level resource managers. In this instance QoS is defined as the fundamental services required by an application to function correctly and in a timely manner as defined by the software requirements. QoS can help runtime adaptivity, something in the domain of classical distributed systems research. But it can also help the applications evolve over their lifetime to handle new requirements or to operate in new environments, issues more in the domain of software engineering but of crucial importance to users and maintainers of distributed systems. Middleware is particularly well suited to provide QoS at an application program's level of abstraction. Also, the abstractions middleware systems offer can often be extended to include a QoS abstraction while still being a coherent abstraction understandable by and useful to the programmer. Distributed object middleware is particularly well suited for this due to its generality in the resources it encapsulates and integrates.

Providing QoS can help applications operate acceptably when usage patterns or available resources vary over a wide spectrum and with little predictability. This can help make the environment appear more predictable to the distributed application layer, and help the applications to adapt when this predictability is impossible to achieve. QoS can also help applications be modifiable in a reasonable amount of time, because their assumptions about the environment are not hard-coded into their application logic, and thus make them less expensive to maintain. Middleware that includes QoS abstractions can enable these things by making an application's assumptions about QoS, such as usage patterns and required resources, explicit while still providing a high-level building block for programmers. Further, QoS-enabled middleware is a high-level building block, which shields distributed applications from the low-level protocols and APIs that ultimately provide QoS. This shielding can be very helpful because these APIs and protocols are very complicated and usually change rapidly compared to the lifetime of many distributed applications. This decoupling of application from lower-level details is thus helpful much in the way that TCP/IP historically has allowed applications and devices to evolve separately. Yet QoS-enabled middleware allows decoupling while providing not only a message stream but with QoS and a high-level abstraction.

9. History of Middleware

The term middleware first appeared in the late 1980s to describe network connection management software, but did not come into widespread use until the mid 1990s, when network technology had achieved sufficient penetration and visibility. By that time middleware had evolved into a much richer set of paradigms and services offered to help make it easier and more manageable to build distributed applications. Concepts similar to today's middleware previously went under the names of network operating systems, distributed operating systems and distributed computing environments.

Cronus was the major first distributed object middleware system, and Clouds and Eden were contemporaries [66]. Birrell and Nelson first developed RPC circa 1982 [66]. Early RPC systems that achieved wide use include those by Sun in its Open Network Computing (ONC) and in Apollo’s Network Computing System (NCS). The Open Software Foundation’s Distributed Computing Environment (DCE) included an RPC that was an adaptation of Apollo’s, provided by Hewlett Packard (acquired Apollo). Quality Objects (QuO) was the first middleware framework to provide general-purpose and extensible quality of service for distributed objects, namely execution verification and timely response for real time systems [3]. Although Quality Object did provide real time type capabilities, there was still no guaranteed scheduling and execution in the context of hard real time systems. TAO was the first major CORBA system to provide quality of service, namely real-time performance. [38][39].

The OMG was formed in 1989, and is presently the largest industry consortium of any kind. The Message Oriented Middleware Association (MOMA) was established in 1993, and MOM became a widely used kind of middleware by the late 1990s. In the late

1990s HTTP became a major building block for various kinds of middleware, due to its pervasive deployment and its ability to get through most firewalls.

The abstraction layer can be either physical or logical. A physical abstraction layer can be utilized, in the sense that the abstraction was implemented in an actual architectural layer. Thus, we create a physical abstraction layer by building a technology-independent virtual machine within which our system components operate.

With a technology abstraction layer, another choice is available. We can also build a logical abstraction layer by designing our system to be technology-independent and then, at the last minute, mapping the design to a specific technology implementation

Using a physical abstraction layer can be advantageous in scenarios where many different technologies need to be supported simultaneously. For example, if I were building a product that I wanted to be able to sell into organizations regardless of whether they used COM, CORBA, or EJB technology, a physical abstraction layer is a good choice. It lets me easily accommodate all of these technologies (and future ones) without affecting the core business logic of my product. This is a key concept in the missile middleware.

However, a physical abstraction layer can be complex to build, and it introduces additional overhead that may affect performance. The reason for this is that in a physical abstraction layer, the execution path used by a particular component is dynamically determined at run time. In contrast, a logical abstraction layer results in a more or less hard-coded translation, and is therefore more efficient.

In most cases, the cost of building the physical abstraction layer is not justified, and the choice of a logical abstraction layer makes more sense. The best way to understand this logical abstraction layer approach is to look at it in the context of the overall system architecture.

G. AN APPROACH TO ABSTRACTION LAYERS – MODEL UNIFICATION

In the following example, originally described by Walker Royce [60], I would like to present a conceptual bridge between software system conceptualization and final representation. It is simplistic, but proves a point. Let us consider a concept system C, a "real world" domain W, and a software system S. We can now view the software development process as the construction of a mapping P that takes concepts and maps them to software elements. An essential quality of P is that it should be faithful, in that events in S map back in a consistent way to events in C, relative to W.

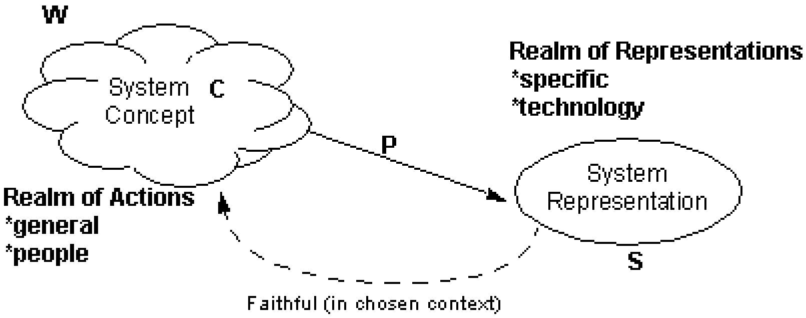


Figure 11. Abstraction Approach

The conceptual system C lives in the realm of actions, which provides two main qualities. First, the system C has generality in that it is defined at a high level of granularity and as such can live simultaneously in multiple contexts within W. Second, C is defined within the context of people, or "real world" actions and events. The software system S lives in the realm of representations, and it too provides two main qualities. One is specificity that is a software system must represent something specific, otherwise it would not be a well-defined representation. The other is that S must be defined within the context of technology, for that is the vehicle for software representations. This is summarized in figure 12.

The required translation from C to S must take place through a process that gradually evolves C from general, people like qualities, to specific, well defined technology-oriented structures. One effective approach to this is to pass the system concepts through a series of intermediate representations (i.e. model layers such as analysis and design) each of which refines the concepts into structures appropriate to the overall qualities for a particular model layer. Although the characteristics of the realms of action and representations are not entirely orthogonal, they do provide a reasonable basis for intermediate representations and thus define model layers as indicated in the diagram below (which is not meant to imply any order in which the layers are traversed or layer size):

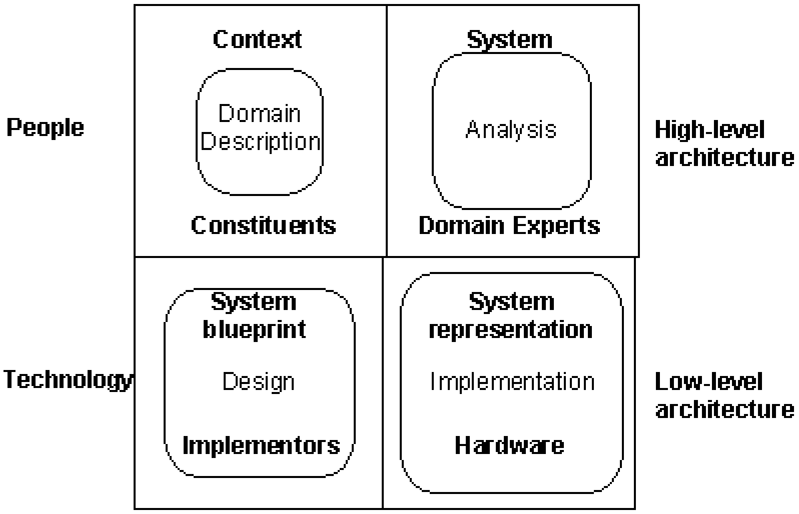


Figure 12. Architecture Context

We find the entity, component, object, and data structure model elements by identifying the forms that exist within each model layer. For example within the Domain Description, the general-people kind of form is an entity of the organization domain, and a specific-technology kind of form is a data structure within the implementation. It is natural to consider components in the analysis of a system and objects within the design. At a higher level of abstraction, closer to the conceptual system, Components capture the high-level architecture. Similarly, objects are closer to the implementation, and hence capture the lower-level architecture.

1. Abstractions

We have set up the framework that integrates various model layers that bridges the gap from C to S in an evolutionary way (although we did not specify a particular evolutionary process). There is one major issue, which has been neglected up this point, and it is vital. For all this to be meaningful, it must actually operate on something meaningful. This implies the need for a structure that we can operate on in a general way that preserves important information and can be easily refined as the structure transition throughout the layers. For this, we introduce the idea of abstraction. An abstraction will serve as the basic "stuff" from which we create models and in a general and flexible way can represent the concepts that arise within our development process.

Quality is a fundamental element of W that represents a value. Qualities may themselves be abstractions, which contain other qualities. Quality values can only be resolved through constraints (even if the constraints are implicit, obvious, or unstated). Adding qualities to an abstraction (or to existing qualities) constitutes a refinement, whence it becomes more specialized. In this way abstractions unify the various model elements. An abstraction may start as a system responsibility that involves various entities. These entities may then be specialized for use in a particular system as components. In turn, components may be represented in software as collections of objects. Finally, each object will be implemented through some manner of data structure.

Although there are a great many interesting issues regarding abstractions, such as elegance and engineering, they can not all be addressed here.

The conceptual system C is literally a collection of concepts. A concept is a difficult thing to work with directly, and thus each concept is represented with abstractions. In our case, let C = (a1, a2, a3, ...) be the set of abstractions that represent the conceptual system. Note that the only assumption being made is that there is a conceptual system. It’s not critical that this representation be complete, consistent, or even well defined. Such issues can be handled incrementally through the lifecycle that constructs the map P.

2. Abstraction evolution

We now culmination our approach to model unification. Since our ultimate goal is to produce S from a faithful mapping P, specificity requires the break down of concepts contained in C into more refined abstractions that have the qualities of software. Abstractions start out general and due to pressures applied at each layer (e.g. domain description, analysis, design) evolve, and the abstraction becomes more specialized through refinement. It is common to break an abstraction up into more specialized abstractions, such as a component decomposing into several objects when moving from analysis to design. This process is called abstraction evolution. Treating all elements in the domain as abstractions allows for a unified approach to modeling. The application of common techniques to describe and operate on abstractions and the model can be adjusted by changing the type of abstraction used to represent a particular construct. For example, an atomic element may start out as an attribute, then become a relationship, then an object, but the essential information is preserved throughout the transitions.

Implicit in the evolutionary process is the notion that only the most useful and fittest abstractions will survive. The names change to reflect the degree of specialization an abstraction has attained and the new qualities it has acquired. An abstraction may stop evolving at any point, but may start again, or influence other abstractions or become extinct. Eventually the set of abstractions (a1, a2, a3, …) will evolve into a set of software abstractions (s1, s3, s3, …) preserving their essential concepts from C, thus encouraging faithfulness, but now having qualities of a software system. The figure 13 depicts the various specializations that an abstraction can undergo through the various model layers.

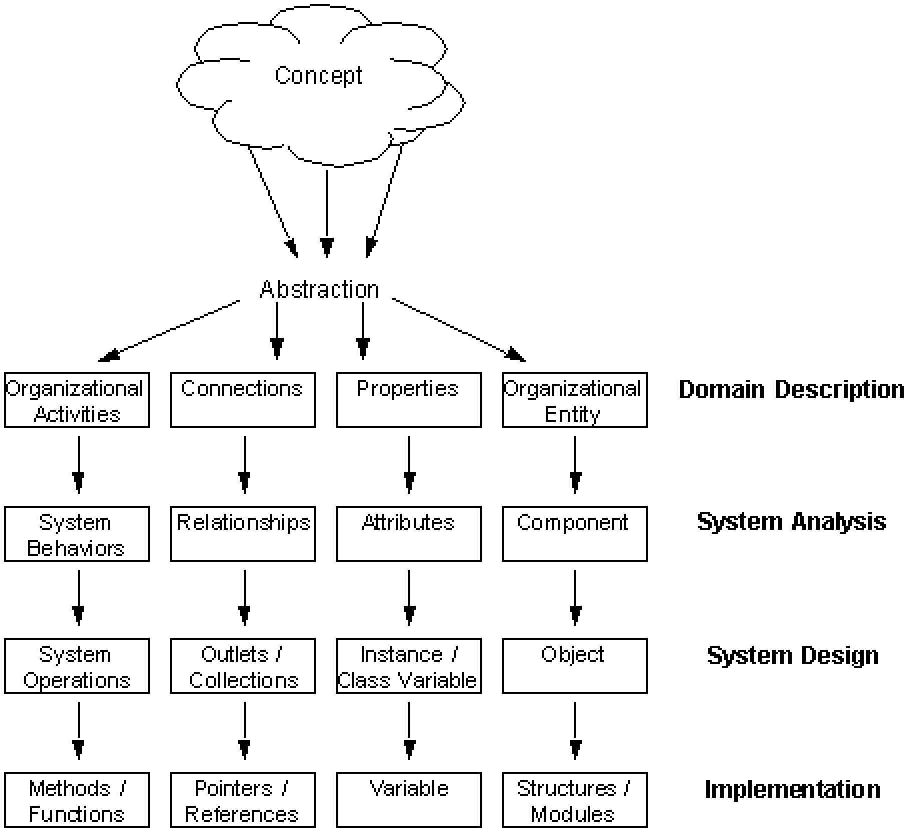


Figure 13. Abstract Evolution

H. THE MISSILE ADVANCED PROCESSSING (MAPS) IPT SOFTWARE ARCHITECTURE – MIDDLEWARE

The middleware approach is a direct answer to, and contribution of this dissertation respective to research question – How can such software architecture abstract the particulars of hardware, and alleviate associated hardware obsolescence. In addition, this approach is analogous to the WSTAWG COE and RT DII COE structure architecture, with two major differences.

1) The middleware and adoption of a real time operating system board support package (BSP), abstracts middleware services, and the functional application components from hardware specific dependencies.

2) While the functional organizational concepts of both WSTAWG and RT DII COE are maintained, services not relevant to the missile domain have been eliminated to reduce overhead, associated induced latencies, and smaller memory footprint.

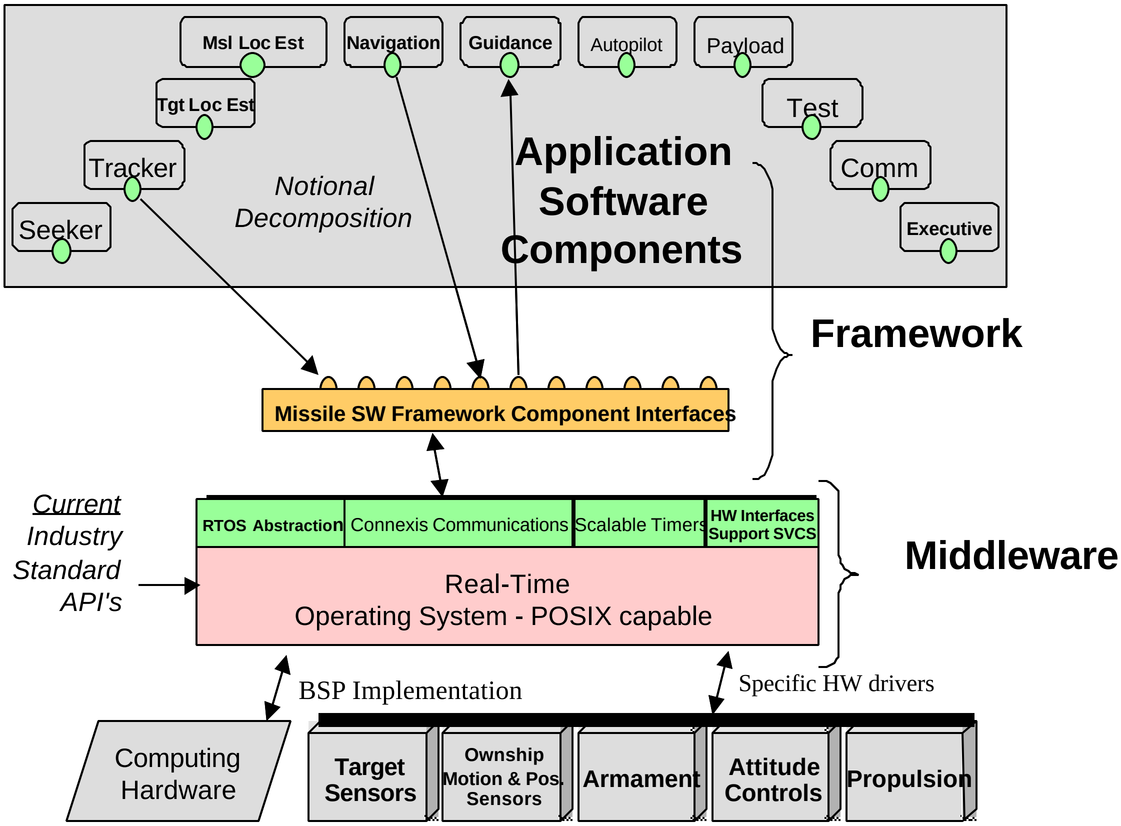


Figure 14. The Missile Advanced Processing (MAPS) IPT Software Architecture – Middleware

1. Middleware, Current Status and Capability

As annotated in figure 14, the hardware abstraction layer is a conscious decision to utilize the board support packages (BSPs) available with the chosen OS - VxWorks, to provide the hardware "porting" mechanism. The alternative is to build our own operating system, which places the project with the prospect of providing life cycle support for the OS as well as the software architecture itself. This would not be a desirable option. For the moment, VxWorks is the commercially available standard OS utilized in hard real time systems. The BSP can be developed in-house to meet current and projected hardware requirements.

OS Abstraction layer - In recognition that change is inevitable, an OS abstraction is provided in the lower middleware layer. This is accomplished via the development tool suite - Rose RT (UML), again a de facto industry standard. The development tools handle migration from one OS to another, which modifies instances of functionality to conform to the chosen OS. API function calls to the OS will be passed through a proxy allowing adaptation to a new OS in the future.

Communications support - Provided by the commercial product Connexis (provided as source code, royalty free distribution with the Rose RT Development Studio. Supports all common OS's).

Connexis provides a transport and protocol independent commercial product which is maintained current with technology by the vendor. The package comes bundled with TCP/IP, and UDP, however, the developer is free to bind any other - either commercial, or homegrown transport and protocol layers. In addition, Connexis provides nodeless communications. While a client - server relationship is supportable, Connexis is not dependent upon this. Many topologies are supported including synchronous, asynchronous, event processing, ORB heterogeneous communications, etc. As a consequence, components of the software system do not have to be co-located. Distribution of functionality can be across the missile or any network (RF, Satellite) meaning processing can be distributed across platforms and systems. The method of interprocessor communication is independent of the functionality of Connexis allowing integration with MPI, Infiniband, or other emerging standards as desired. Connexis provides interprocess communications support. This enforced tight integration – cohesion of the software, but loose coupling. Loose coupling refers to the modular characteristics of the function components above the API. Having loose coupling allows individual components to be removed from a given system, without also extracting a convoluted spaghetti mess of code from the rest of the original system. There are no explicit dependencies. Modular code reuse becomes a reality. Connexis provides support for distributed heterogeneous processing and parallel processing, and standardized interface for communication between processes

Scalable Timers, is another feature of the middleware services, which allow for multiple, independently referenced timers to be utilized within the context of the software architecture. The current implementation allows granularity to 1 nanosecond. The independence, and scalability of the timers, which can be instantiated as needed, provides the synchronization mechanism for the entire software system. This allows the tactical software to work in a “real”, “simulated”, or “simulation” environment.

The Application Program Interface (API) provides the “glue” to begin integration. The next step is to begin development / integration of real missile software to define the standardized interfaces and groupings.

V. REALIZATION OF A HARD REAL TIME EMBEDDED SOFTWARE ARCHITECTURE

A. SYSTEM GENERALIZATIONS

Previous descriptions, in previous chapters, of the systematic descriptions of hard real time software architecture have been at the historical and very generalized level. The missile software represents hard real time in the most critical sense utilized today. There exists no constraints entailing more time critical, and sensitive execution anywhere. The research conducted over the past several years has resulted in a practical approach to such system realization, which is working, and is being implemented in several missile systems.

The specific contribution to the advancement of Software Engineering is the engineering design, development, implementation, and demonstration of a software architecture providing a hardware abstraction, middleware common services, and a polymorphic application program interface, component based interchangeable functional blocks, capable of supporting the hard real time constraints of a modern missile system. This is the first software architecture designed specifically to meet the time sensitive requirements of missiles. Additional benefits of the proposed software architecture include portable software code; only the board support package of the operating system need be modified for a new hardware representation, the remainder of the software code is simply recompiled for the new target hardware. The communications and processor inter-processor communications is handled by a CORBA like product, Conexis, which provides such communications through independent transport mechanisms and protocols. The Conexis products itself functions via the services provided by the middleware, while maintaining cross platform compile capability, and hardware abstraction. Presently, Conexis operates in a missile hard real time context, and is compatible with much of the functionality of Real Time CORBA definitions, allowing future insertion of a full up real time CORBA OBR, if and when such services become available, and desirable.

B. USAGE OF PATTERNS IN THE HARD REAL TIME SOFTWARE ARCHITECTURE

During the course of this research, a pattern of software interaction has emerged which articulates the interfaces and design considerations necessary in realizing a workable architecture for missile systems.

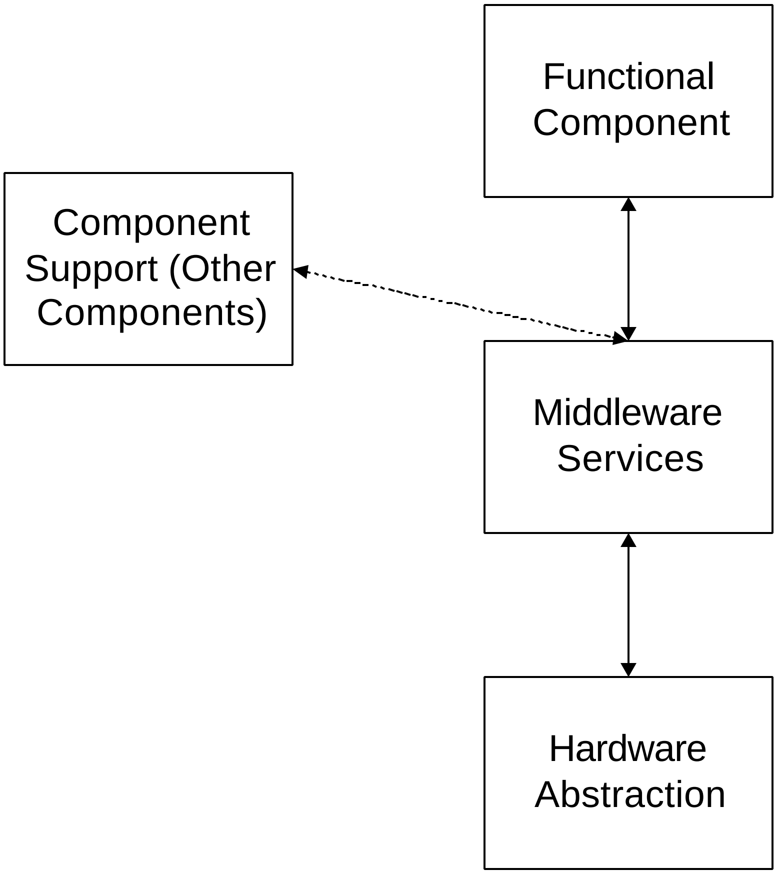


Figure 15. Software Architecture Pattern

As shown in figure 15, the pattern associated with the hard real time embedded software architecture consists of three major pieces, and a supporting functional component. At the lowest level, there exists a hardware abstraction layer, which provides the realization of the specific instance of hardware in a given circumstance. This realization is accomplished via the operating system Board Support Package (BSP).

The middleware services layer provide the glue binding the software to the operating system, while providing abstraction from the operating system, allowing future insertion of alternative operating systems. In addition, the middleware provides inter and intra process communication, multiple independent timers instantiations with independent reference points, and a polymorphic Application Program Interface (API) to which the functional components of the missile system bind.

The functional components define the modular areas of functionality within the particular instance of a missile software design, and the interface specification to the middleware layer, enforcing a tight integration / loose coupling of functions and services within an overall abstracted software architecture which directly supports technology insertion, enhancement, and porting of software to new hardware instances with a minimum of effort. A new hardware instance porting of the software code involves defining the BSP to the new hardware instance, compiling the operating system to the new hardware architecture, then cross compiling the remaining software system code to the new hardware instance.

The hard real time software architecture presented in this dissertation is the first, and only, working architecture supporting the hard real time requirements of missile systems in existance today.

C. REALIZATION OF GENERAL SOFTWARE LAYERS

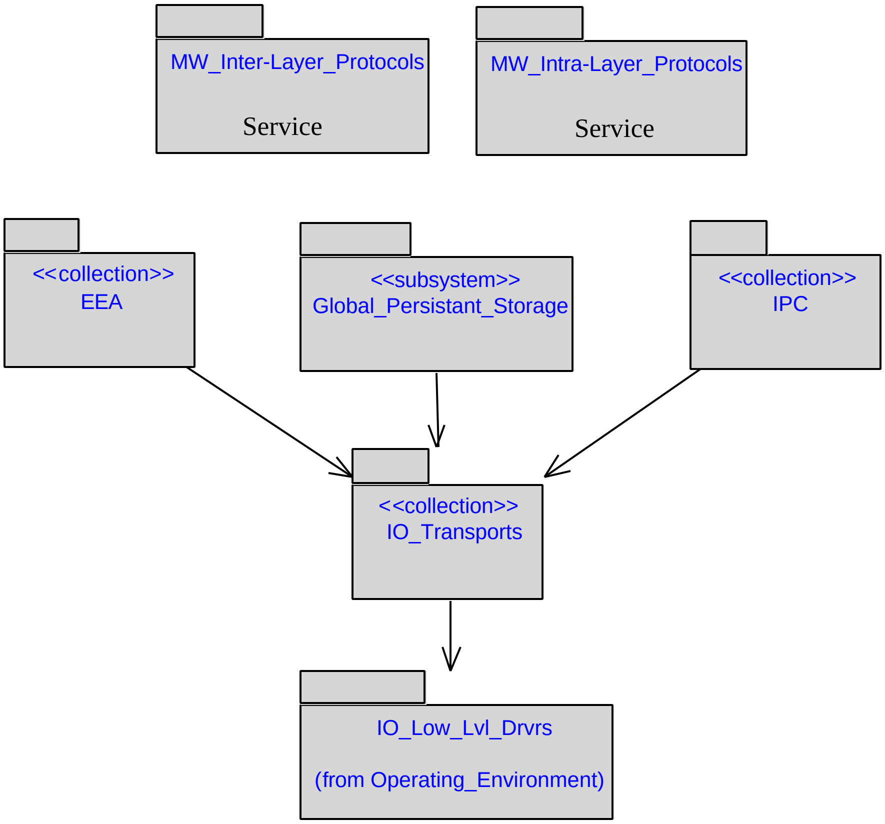


Figure 16. Operating System Abstraction Instance

Beginning at the lowest level, there is a desire to utilize the standardized services of a commercial operating system (OS), in this case VxWorks Real Time (RT), by WindRiver Systems. The ideal is to abstract the OS such that while taking advantage of the services provided, no dependencies arise which would prevent switching to another OS in the future. The primary advantage of a commercial OS is the Board Support Package (BSP), which represents the specific instances of OS bonding to a particular hardware configuration. Another advantage is that the utilization of commercial products alleviates development costs, providing the commercial product delivers and maintains required capability throughout the lifecycle of a given product.

In the case of the example given in Figure 16, the abstraction is from OS dependent hardware specific drivers, via an abstract transport information layer, which in turn communicates with the facilities layers contained within the lower levels of the software middleware layer.

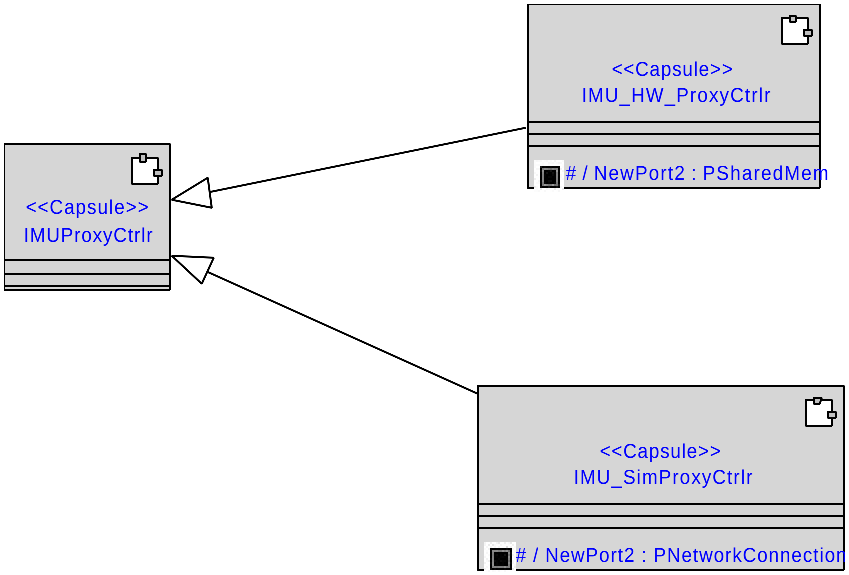


Figure 17. Inertial Measurement Unit (IMU) from OS abstraction layer to IMU Processing control

The Inertial Measurement Unit (IMU), figure 17, represents a part of the Navigation System, however, when dealing with missile systems, it is considered a standard service, and therefore located within the services layer of the middleware. In actuality, the IMU can be realized by any one of several sources of information, including live sensors, network relays, and actual internal dead reckoning input to the missile navigation. Either way the IMU service is an interface bridging the physical hardware manifestation, to the software within navigation. The instance of the IMU_HW_ProxyCtrlr and IMU_SimProxyCtrlr are the information receptors connecting the OS abstraction transport, mentioned previously, essentially an information “pipe” to the instance of software, via an abstraction of the OS / hardware, allowing this service’s reusability through a new OS binding, via a BSP, to a new hardware instance. As might be noticed in figure 16, there are two distinct inputs to the IMU processing pipeline. These input paths, which are common in the services layer of the middleware, provide an externally controlled choice of input that enables the usage of the software, and hence the entire software system in either a simulation, or tactical environment. This capability is a desired functionality within a missile to provide the capability to combine real operational hardware, software in the loop, and simulated components to aid in development, facilitate time to delivery, and provide for “what if” excursions, aiding in the imagineering of new products.

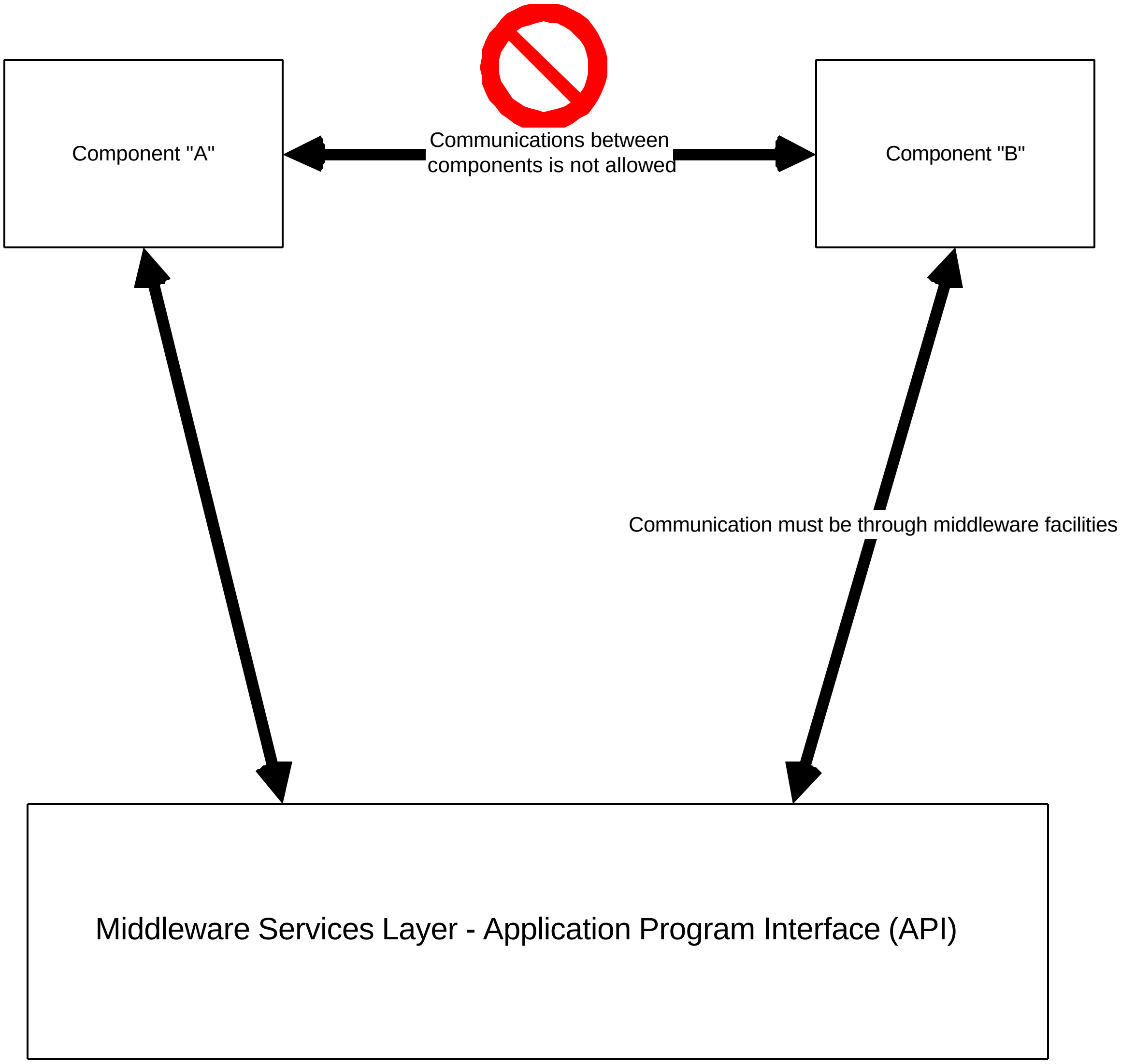

Component "A"
Communications between components is not allowed
Component "B"
Communication must be through middleware facilities
Middleware Services Layer - Application Program Interface (API)

Figure 18. Reflective communications

Interprocess communications must be tightly integrated, but loosely coupled. The primary reason is to enforce absence of direct intimate dependencies between components. By channeling all interprocess communication through an intermediary, it is possible to allow such tight integration without tight interdependencies, thus allowing components to exist anywhere within a given "system", regardless of geographic location. This approach has properties similar to object oriented containers, and the process interactions common to CORBA, but without the overhead of CORBA, although CORBA can be used as the underlying communication mechanism without binding dependencies inherent in object oriented encapsulation.

Another aspect of interprocess communications is interprocessor communications. It is normal within the context of missile embedded systems to utilize much more than one processor. In addition, the multiple processors may be of different types (heterogeneous). While it would be desirable to utilize real time CORBA's such as ACE-TAO, or ACE-ZEN, timing benchmarks run in 1999 indicate that none of the existing object request broker (ORB) implementations provides adequate response time to have been considered at that point in time.

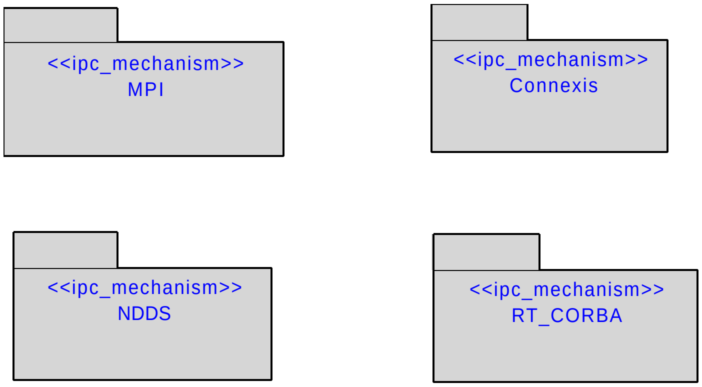


Figure 19. Interprocessor communications methods

Communications between components / processes, often transposes processor boundaries. As illustrated in figure 19, it is necessary for such communications methods to be hidden from the individual components. To that end, within the middleware services layer, mechanisms exist to allow communications between components regardless of physical location. This effort chose the Connexis interprocessor communications package. Connexis is part of the Rational Rose Real Time Development Studio, no additional cost, and provides the required functionality for missile systems. In addition, the source code is provided, allowing cross compilation to any defined hardware architecture. Likewise, Real Time CORBA, Message Passing Interface (MPI), Inifinband, and RapidIO, can be substituted for the Connexis product since in keeping with the philosophy of this research project, the interface to the interprocessor communications package is abstracted, providing for tight integration, loose coupling, and straightforward substitution of major components.

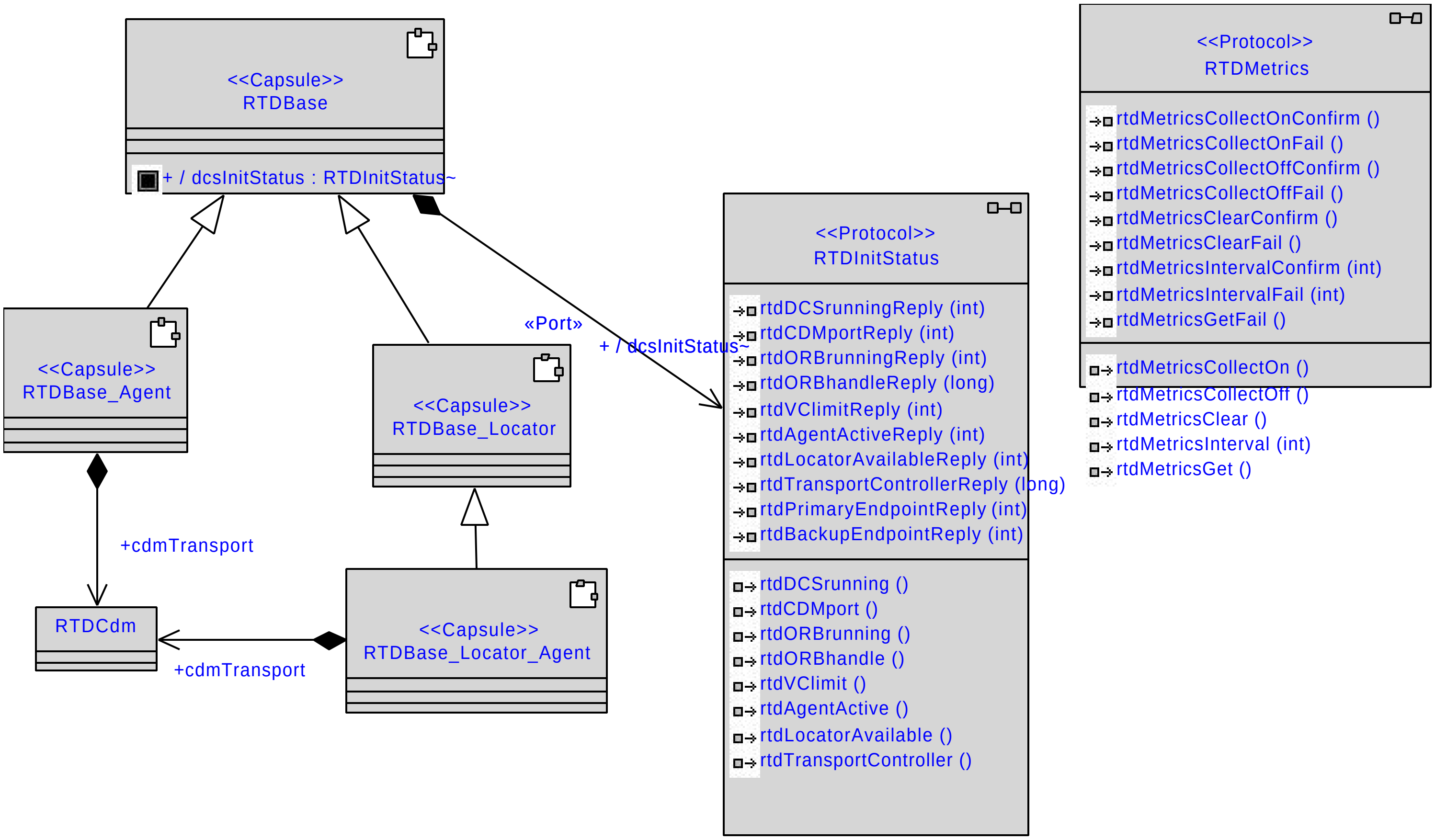


Figure 20. Connexis Implementation – Interprocess / Interprocessor Connectivity

Key to successfully integrating an interprocess / processor communications package, while allowing future substitution of the package, is the creation of an abstract layer which performs the necessary communications patterns, while not tightly coupling with the communications package itself. Figure 20 exemplifies the simplicity of abstraction enabling additional functionality to be created, on the fly, while remaining backward compatible with previous instances of the Connexis implementation.

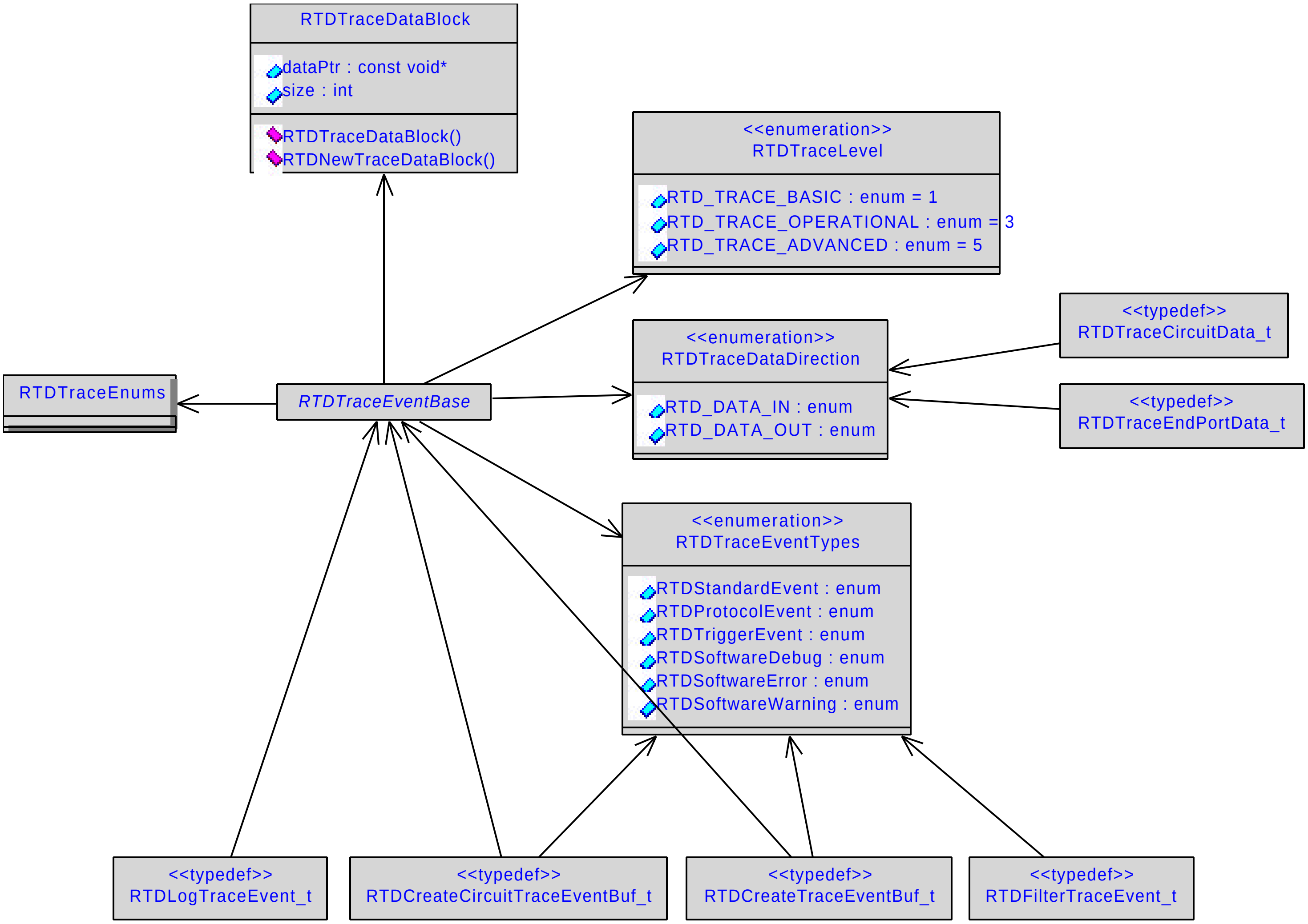


Figure 21. Real Time Data Trace Dependencies - Interfaces

Now that the communications mechanisms are defined, along with the required behavior patterns, it is necessary to specify data relationships, and trace methods, figure 21. The advancement of the state of the art is the realization of architecture supporting hard real time embedded software. There are no previous architectures, as the domain of hard real time is only now being explored, hence, this is the first of it's kind. A trace method is a pattern of data dependencies relative to the context in which data is realized in a given instance of system operation. In essence data trace dependencies ensure the integrity, and validity of data, allowing transformation to information, providing the point of reference for the system, in this case a missile, to act on knowledge.

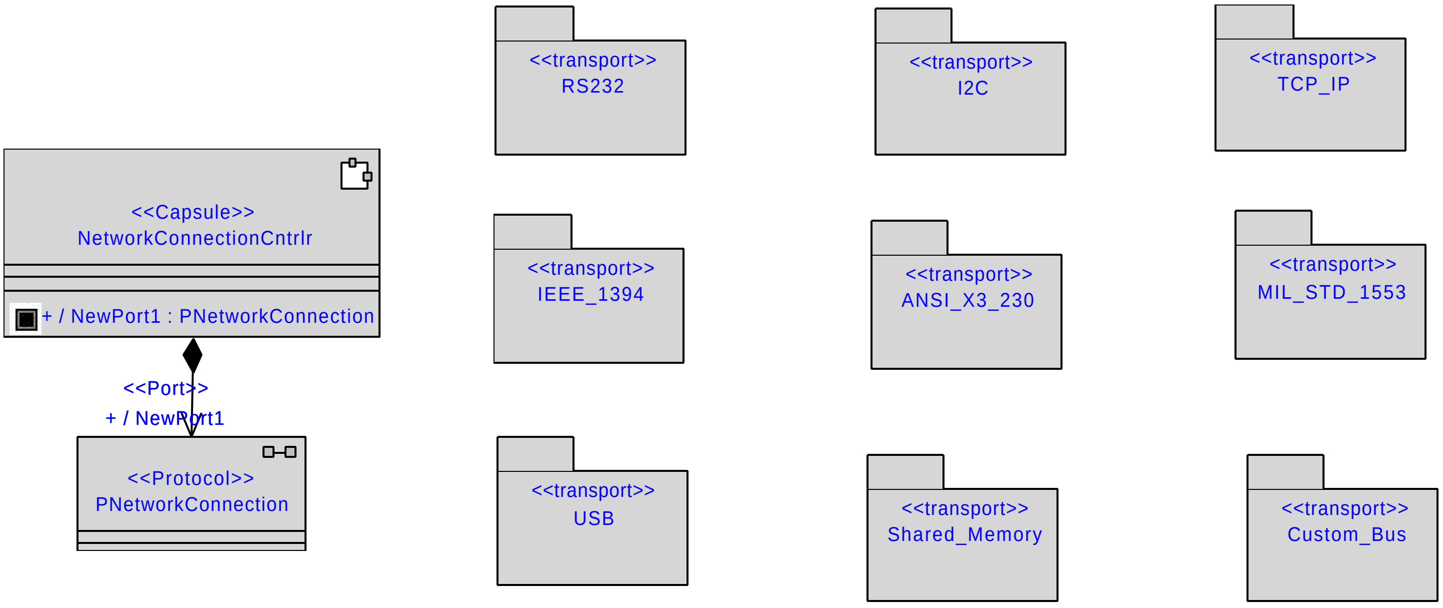


Figure 22. Protocol Instances

As part of the communications services, the transport and protocol implementations, figure 22, are independent of the software system definition. Any one or more methods may be invoked for communications. In addition, the definition of new protocol / transport standards is simply a matter of defining the interface specifics, making for an extremely adaptable implementation of physical communications properties. Furthermore, the transport services are independent of the rest of the system, so that their implementation can be reused in many different applications. As systems become more interconnected, NET appliances are one such example, the requirements of synchronization, and distributed processing will dictate hard or hard like software solutions.

Timing and synchronization is critical for any component-based software system to function properly, particularly in the context of hard real time systems.

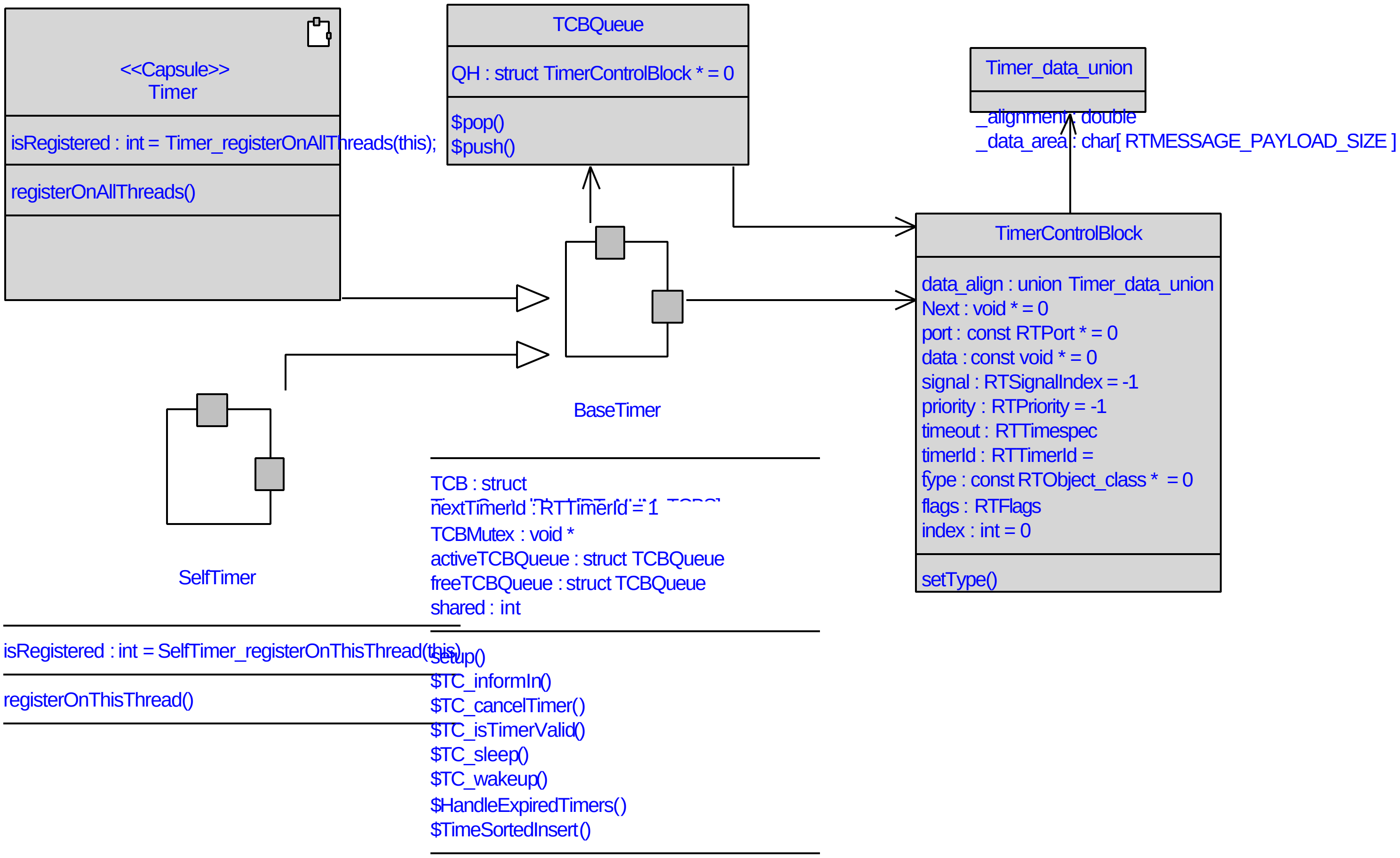


Figure 23. Timer Dependencies (TimerPackage)

Timers required for this research effort must have three characteristics. First each timer must operate independently of one another, second, each timer must have variable scalability (from microseconds to milliseconds), and finally, the reference source for each timer must be able to be independent. Independent operation refers to the ability of each timer to maintain its precision, without dependency on any other timer. The scalability of timers refers to the timing increments required by of each timer's needs. For example an RF sweep may operate at a frequency of 20Hz, while updates to target recognition may operate at a frequency of 700 microseconds respective of each line of resolution in the capture image. Each line of resolution is buffered in the radar hardware until a packet of information is available. In general the packet of information to be offloaded for processing will be multiple instances, constituting a piece of, or entire "picture" (multiple lines). Keep in mind this refers to a device driver, and is not relying upon the middleware services to interface directly with the radar hardware. The reference source for timing must be adjustable for each timer, as dictated by the timer's requirements. For example a radar sweep having a period of 500 milliseconds must be synchronized to the processing of the radar, not some other component for data transfer synchronization. Another option provided, and discussed briefly earlier is the ability of components within the software system to operate, that is receive / provide stimulus, to both tactical and simulated hardware components. The timer must be able to accept a timing stimulus from either source.

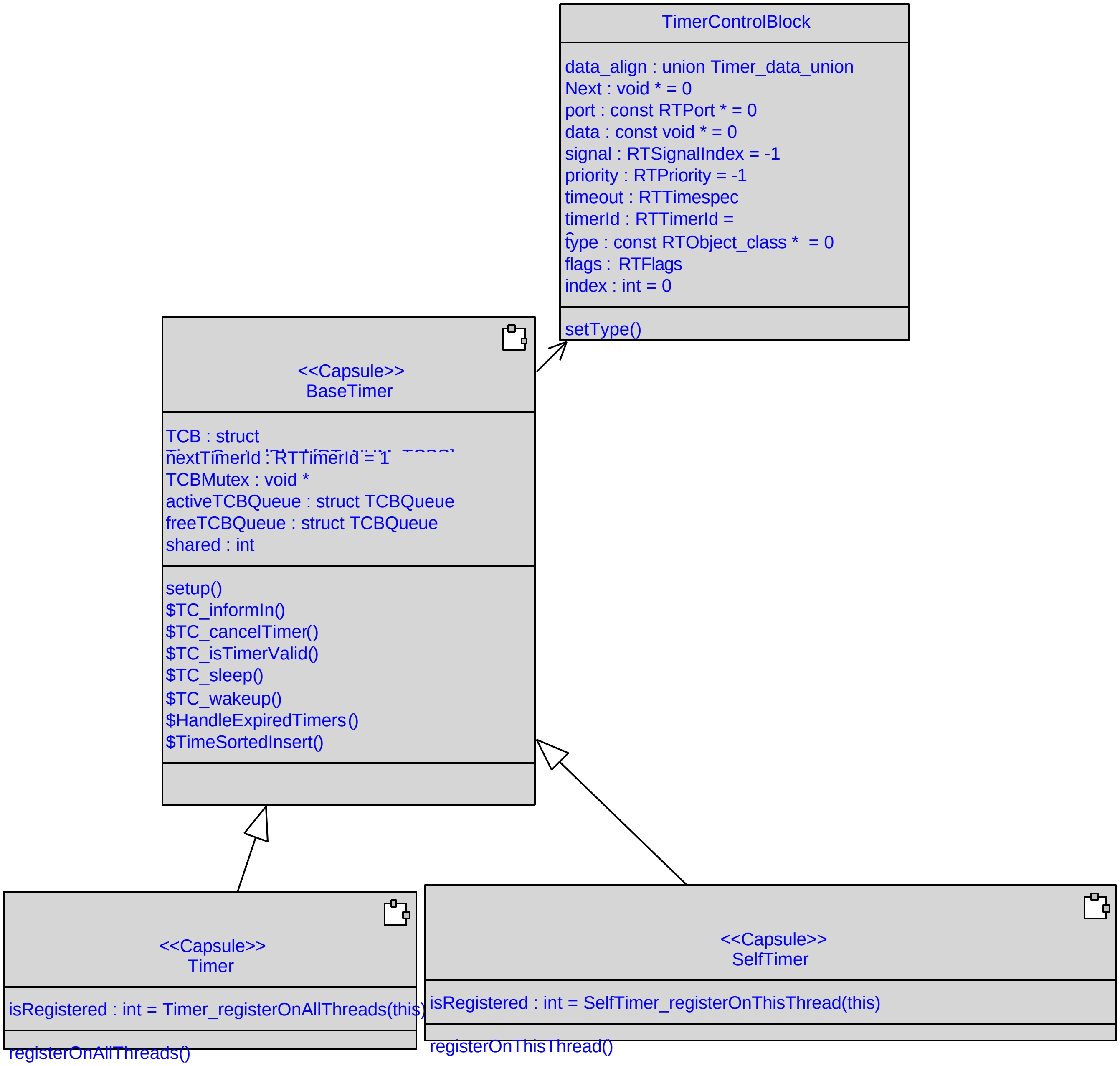


Figure 24. Timer Class Specification

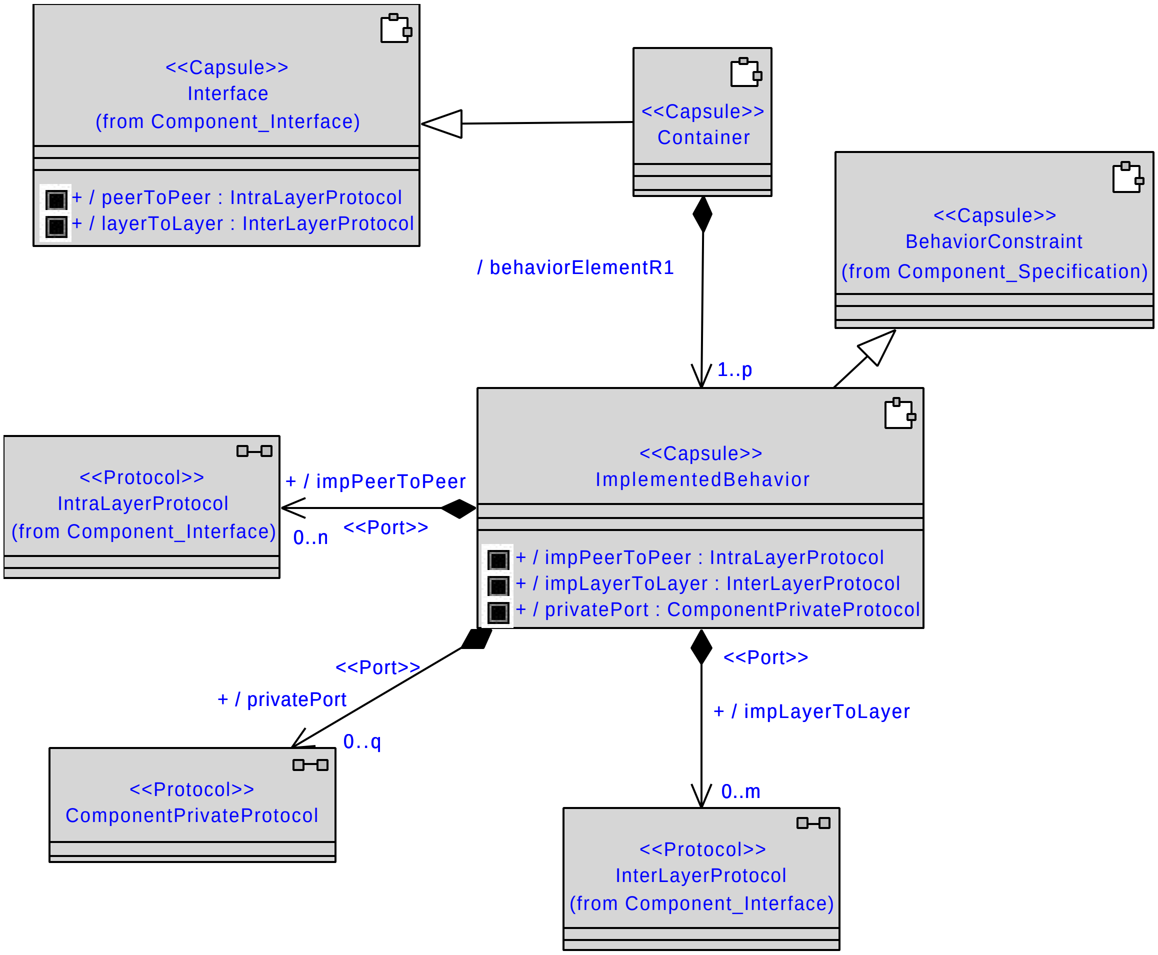


Figure 25. Layer Pattern Class – Component Realization

Services defined within the middleware layer are to support the components above the Application Program Interface (API) layer. Through the standardized definition of services, and means to exchange information external to the individual component, a pattern of behavior can be established allowing the definition of a standardized approach to be utilized in component realization. The template permits a higher level abstraction of the generalized interface definition, without making assumptions about the specifics related to the individual component. By means of this approach, the component based interchangeable logic remains intact, without placing undo constraints on the components.

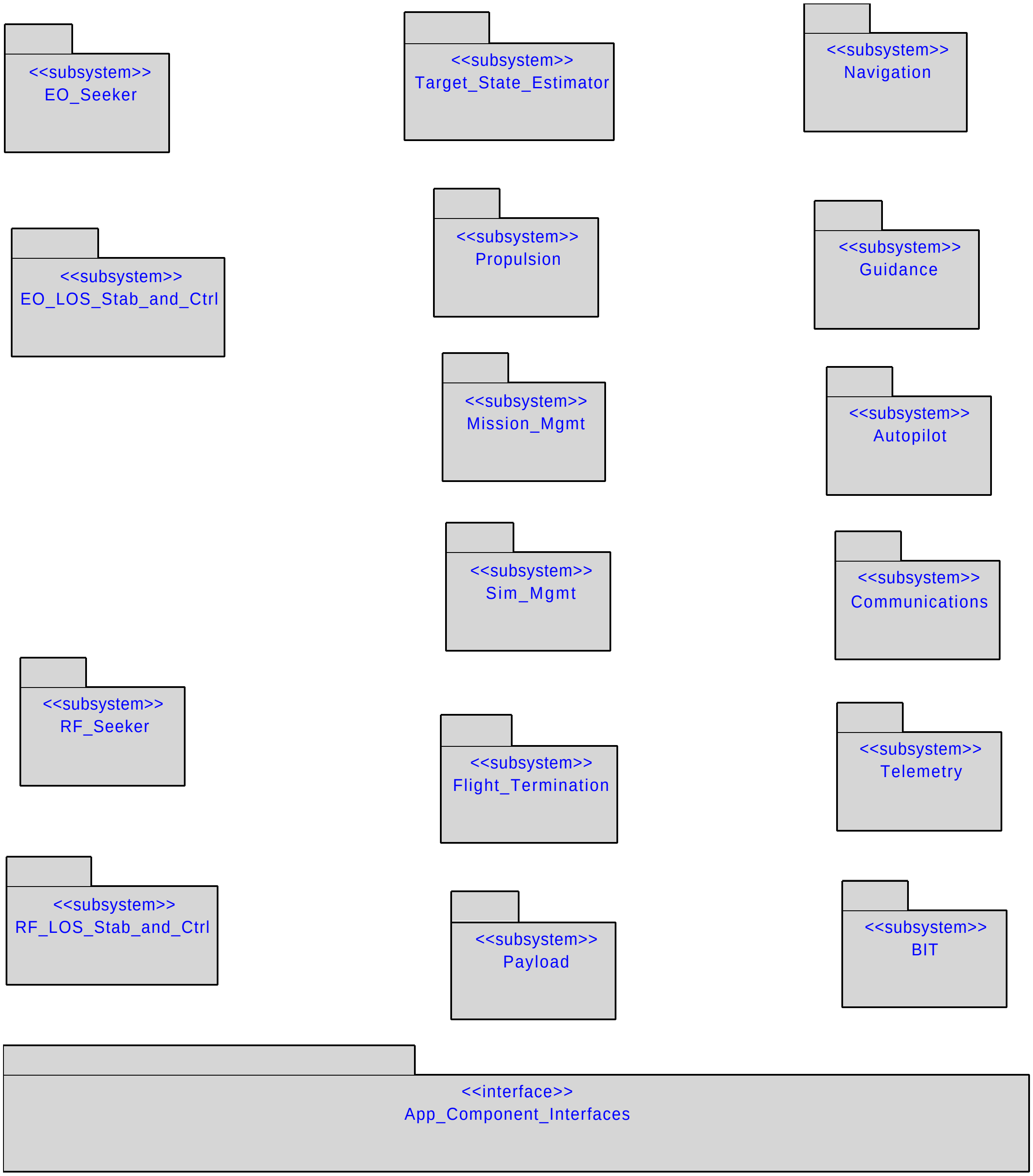


Figure 26. The API Interface and Missile Generalized Components

Definition: *An API allows programmers to access the functionality of a pre-built software module through well-defined data structures and subroutine calls. Although programmers often define APIs for private internal code, network APIs typically are the public entry points to libraries that hide low-level details of computer operations.* [61]

The associative relationship between middleware services and the functional components, figure 25, above the API follow the Microkernel, or Layer Pattern as defined by Bruce Douglass [28]. The lowest level in the hierarchy provides services directly to, and via the Real Time Operating System (RTOS), with the underlying hardware instance, while the upper layers contain high level applications domains, or in this case, components. The primary advantage of the Layer Pattern is the componentization (isolation) of functional parts of the overall software system, which greatly increases portability, manageability, and reusability within complex software systems. The major disadvantage is loss of performance in such software designs.

The loss of performance has been mitigated, to a degree, through use of faster processors, the Rose Real Time target environment code generation, and optimized cross compiler technology. The purpose of this research is not to analyze specific tools, but rather the design and implementation of a hard real time embedded software system. As such I am a consumer of these technologies, not the producer. Suffice to say that faster processors will process the associated overhead code required for Layered Patterns quicker, the Rose RT development environment has the capability to generate pseudo optimized code for a target hardware environment, and cross compilers allow development in a common, less expensive environment – Windows NT / 2000, and optimized binary code generation directly to the target hardware environment. These topics, in and of themselves could be the topic of an entire dissertation, but will not be discussed further here.

VI. CONCLUSION

A . SUMMARY, CONTRIBUTIONS OF THE DISSERTATION

In the presentation of information in this dissertation, all the research questions and contributions have been answered. Briefly I will review each research question, and contribution, most of which is highlighted in the body of the dissertation.

1. Contributions

Contribution 1

The first contribution of this dissertation is the association of current software technology standards toward the solving of the hard real time software architecture constraints such as componentization, layered abstraction, and time synchronization. Although the toolset for such a design and implementation is not complete, or mature, it is possible to address this problem, successfully, while keeping the design open to allow insertion of new technology and methods in the future.

Current industry practice with respect to hard real time embedded software is one of hand crafted solutions addressing a specific problem. Practices do not follow software-engineering practices or technology advances made over the past twenty years. Object oriented solutions, along with structured approaches to software development are, for the most part, non-existent within the hard real time domain. Recent advances in processor technology, mainly speed make practical a structure, layered approach to software development as delineated in this dissertation. The realization that such possibilities can be achieved, within the context of guaranteed execution, and cost effective and maintainable component based software architecture, is a primary contribution of this dissertation. Standards integration, primarily the utilization of UML, make possible a structured engineering design capable of handling software system complexities in a measurable, quantifiable, and maintainable context. Introduction of design aid tools, along with automatic code generation capabilities, such as the Rational Rose Real Time Development Studio, provide an integrated package for such design, combined with documentation, error checking, configuration management, and the realization that model begin to replace the software code as the reference in system design, assembly, and integration.

Contribution 2

A second and most important contribution is the development of a functional hard real time embedded missile software architecture. Through utilizing examples of soft real time architectures, such as the Defense Information Infrastructure Common Operating Environment (DII COE), and others, the necessary functionality for a missile system, comprising a well understood domain [62], has been transposed to a layered component based approach, entailing the necessary services, and not the unnecessary overhead associated with soft real time domain.

The end product and contribution of this dissertation is the realization of a functional embedded hard real time software architecture to be utilized within the missile domain. This is the first such proposed software architecture which provides the functionality, abstraction from hardware implementations, and componentization meeting missile requirements, while maintaining compatibility with structure software architectures in other than hard real time. While it is recognized that compatibility has many forms, here it applies to the capability to exchange information in either synchronous or asynchronous form. Attempts have been made to provide a layered software architecture for soft real time systems, such as RT DII COE, and a pseudo-hard real time domain, WSTAWG, but to date no software architecture has been proposed to define, and implement the necessary capabilities to satisfy the requirements of the missile domain.

Contribution 3

The third contribution of this dissertation is the realization that the benefits related to Moore’s Law, processing capability doubling every eighteen months, or less, can be utilized to advantage when challenging previously unthinkable problems. Embedded processing solutions can address a number of areas of research ranging from communications, to weapons systems, to medical technology. We have the know how as to what needs to be accomplished, but until very recently did not have the computational horsepower to effectively implement the solution in a cost effective manner. It is hoped, that by example, old solutions and problems thought beyond practicality, be re-examined.

“Tradition unhampered by progress” all too often this seems to be the *defacto* standard in both industry and government. In some ways this might make sense, if something isn’t broke, don’t fix it. On the other hand, when that something is devouring large quantities of people power, schedule, and budget to maintain and enhance, it is time to re-examine the problem, specifically looking for new solutions to old problems. This dissertation has examined the domain of an embedded hard real time software architecture for missile application. The realization of such architecture within the context of the missile domain will bring order to chaos. Additional cost savings through reduced schedule, accuracy of the software through validation, and re-validation in successive implementations will provide a more reliable product, and significant real dollar cost savings.

B. USABILITY AND EXTENSIBILITY

The previously demonstrated design has successfully provided hard real time embedded software architecture capable of executing within the strict time constraints of a missile system. Work is ongoing implementing this architecture into real missile systems, which will constitute the next generation missile systems being fielded in the near future. This project is unique, and pushes the realm of software technology, in that it combines modern software practices, in a modular component based schema, utilizing layered pattern techniques, to produce a flexible, adaptable, and interchangeable component based architecture having a clear migration path into the future. Such approaches have been accomplished in the past for soft real time systems, such as C4I, but the differences between the system constraints of a hard real time versus a soft real time system, have been to date incompatible.

Initial benchmark test of the Middleware, as a part of the overall Missile Hard Real Time Embedded Software Architecture has been quite favorable. Utilizing the Rational Connexis inter – intra process communication software in an unoptimized state has yielded timing latencies in the range of 180 – 210 milliseconds, well within acceptable limits for most missile applications (I am not allowed to site the sources of this information). Recent benchmark data provided by Doug Schmidt (DARPA – ACE TAO fame) has demonstrated latencies of 110 – 120 milliseconds for the latest release of the ACE RT ORB. The intention is to benchmark the new ACE TAO within the missile architecture, and if the numbers hold, and the memory footprint is acceptable, replace Connexis with ACE. The modular construction of the middleware allows easy replacement of components. In addition the communications substructure is independent of transport / protocol, allowing easy adoption to whatever method of invocation is required to meet the system requirements.

C. FUTURE CONSIDERATION

This dissertation constitutes a conglomerate of current research, recombined to produce a unique workable solution to a specialized domain within hard real time systems. While such efforts as DII COE RT are organizing, through committee, an architectural approach to these problems in soft real time software intensive systems, there exists no tangible, demonstrable, software architecture capable of accommodating the requirements of hard real time embedded systems such as that required in missile system, flight control software, and space flight control. The requirements of speed, coupled with assured Quality of Service (QOS) have dictated other, more instance related approaches to problem solutions, with little or no hope related to code reuse, system extensibility, and life cycle cost of ownership. Today thanks in large part to the benefits realized from Moore’s law, the hardware, especially processors, memory, and bridges are becoming flexible, and responsive enough to allow a more structured approach to hard real time software architectures.

In the realm of dependable computing, work is necessary to produce a fault tolerant, QoS capability within software architectures related to hard real time embedded systems. While all normal measures are taken to ensure accuracy and reliability of software operations, there exists no timely, graceful way to handle exceptions, and unforeseen errors within hard real time software. Such measures are necessary to prevent catastrophic failure in the event of software system error, and in the case of missiles, probable loss of life.

Systems integration is of major importance in the consideration of large scale DoD information processing. To that end, little effort has been made, or proposed, to bridge the gap between non-real time, soft real time, and hard real time systems. Information exchange between such systems is only a part of the solution. The need to address knowledge sharing, and overall systems synchronization to enable coordinated systems interaction is significantly more than just sharing information. Such coupling of systems would provide smart, integrated solutions to problems utilizing the most effective combination of resources in a coordinated fashion, maximizing positive results.

VII. APPENDIX A

A HARD REAL TIME EMBEDDED SYSTEM REALIZATION

Appendix A is proprietary, competition sensitive, and not included in the main body of the dissertation.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. *“Lightweight remote procedure call”,* Association for Computing Machinery (ACM) Transactions on Computer Systems, 8(1):37-55, February 1990.

2. B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gün Sirer. *“Spin - an extensible microkernel for application-specific operating system services”,* University of Washington Computer Science and Engineering Technical Report 94-03-03, February 1994.

3. J.B. Chen, A. Borg, and N.P. Jouppi. *“A simulation-based study of TLB performance”,* In Proc. 19th Annual Intl. Symposium on Computer Architecture, pages 114-123. ACM SIGARCH, IEEE Computer Society, May 1992.

4. D.R. Cheriton. *“The V distributed system”*, Comm. ACM, 31(3):314-333, March 1988.

5. D.R. Cheriton, H. Goosen, and P. Boyle. *“ParaDiGM: A highly scalable shared-memory multi-computer architecture”*, IEEE Computer, 24(2), February 1991.

6. D.R. Cheriton, H. Goosen, and P. Machanick. *“Restructuring a parallel simulation to improve shared memory multiprocessor cache behavior: A first experience”,* Shared Memory Multiprocessor Symposium, pages 23-31. ACM, April 1991.

7. D.R. Cheriton and R. Kutter. *“Optimizing memory-based messaging for scalable shared memory multiprocessor architectures”,* Stanford Computer Science Technical Report CS-93-123, December 1993.

8. D.R. Cheriton, G.R. Whitehead, and E.W. Sznyter. *“Binary emulation of UNIX using the V kernel”,* In USENIX Summer Conference. USENIX, June 1990.

9. D.R. Engler, M.F. Kaashoek, and J.W. O'Toole Jr. *“The operating system kernel as a secure programmable machine”*, Proceedings of the ACM European SIGOPS Workshop, September 1994.

10. A.C. Bomberger et al. *“The KeyKOS nanokernel architecture”,* Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures, USENIX, April 1992.

11. Black et al. *“Translation lookaside consistency: A software approach”,* Proc. 17th Int. Symp. on Computer Architecture, pages 113-122, April 1989.

12. J. Liedtke et al. *"Two years of experience with a micro-kernel based os"*, Operating Systems Review, 25(2):57-62, 1991.

13. K. Anderson et al. *"Tools for the development of application-specific virtual memory management"*, OOPSLA, 1993.

14. M. Rozier et al. *"Overview of the CHORUS distributed operating system"*, Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures, USENIX, April 1992.

15. R. Rashid et al. *"Machine-independent virtual memory management for paged unitprocessor and multiprocessor architectures"*, IEEE Trans Comput., 37(8):896-908, August 1988.

16. K. Harty and D.R. Cheriton. *"Application-controlled physical memory using external page cache management"*, ASPLOS, pages 187-197. ACM, October 1992.

17. J. Kearns and S. DeFazio. *"Diversity in database reference behavior"*, Performance Evaluation Review, 1989.

18. Massalin and C. Pu. *"A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91"*, Computer Science Department, Columbia University, October 1991.

19. R. Rashid and D. Goluv. *"UNIX as an application process"*, USENIX Summer Conference. Usenix, June 1990.

20. M. Schroeder, D. Clark, and J. Saltzer. *"The MULTICS kernel design project"*, Proceedings of the 6th Symposium on Operating Systems Principles, pages 43-56. ACM, November 1977.

21. M. Young et al. *"The duality of memory and communication in the implementation of a multiprocessor operating system"*, In 11th Symp. on Operating Systems Principles. ACM, November 1987.

22. P. Bernstein. "Middleware: *"A Model for Distributed System Services."* , Communications of the Association for Computing Machinery (ACM), 39:2, February 1996, 86–98.

23. A. Campbell, G. Coulson, and M. Kounavis. *"Managing Complexity: Middleware Explained."* IT Professional, IEEE Computer Society, 1:5, September/October 1999, 22–28.

24. J. Zinky, D. Bakken, and R. Schantz. *"Architectural Support for Quality of Service for CORBA Objects"*, Theory and Practice of Object Systems, 3:1, April 1997.

25. Schmidt, D. Levine, and S. Mungee. *"The Design of the TAO Real-Time Object Request Broker."* ComputerCommunications, Elsevier Science, 21:4, April, 1998.

26. P. Verissimo and L. Rodrigues. "*Distributed Systems for System Architects*", Kluwer Academic Press, 2001.

27. Bollella, *"Priorities Supporting Real-Time Computing Within General-Purpose Operating Systems"*, Ph.D. Dissertation 1997, University of North Carolina at Chapel Hill, Department of Computer Science, December, 1997

28. B. Douglass, *"Real-Time UML Second Edition, Developing Efficient Objects for Embedded Systems",* Addison-Wesley, 2000.

29. M. Fowler, K. Scott, *"UML Distilled, Second Edition, A Brief Guide to the Standard Object Modeling Language",* Addison-Wesley, 2000.

30. Gomaa, *"Designing Concurrent, Distributed, and Real-Time Applications with UML",* Addison-Wesley, 2000.

31. B. Meyer, *"What to Compose*", Beyond Objects, Jan. 2000.

32. Jacobson, G Booch., J. Rumbaugh, *"The Unified Software Development Process",* Addison-Wesley, 1999.

33. J. McHugh, *"Binge and Purge Now we know: How ERP software's promise died -- and who killed it***",** eCompany Now, http://www.ecompany.com/articles/mag/0,1640,6580,00.html

34. M. Cantor, "*UML and Communication through the Life Cycle*", http://www.sdmagazine.com/documents/s=815/sdm9904uml4/sdm9904uml4.htm.

35. D. DSouza, Kinetium, "Model-Driven Architecture and Integration Opportunities and Challenges", March 2, 2001 draft paper.

36. Hill, Robert, "*Improving Linux Real Time Support: Scheduling, IO Subsystem, and Network Quality of Service Integration*", Master's Degree Thesis, University of Kansas, 1998.

37. Lt. Col. Lucie M.J. Robillard, et. al. , "*Extending the DII COE for Real-Time*", CROSSTALK The Journal of Defense Software Engineering, September 1999, volume 12, number 9, pp 6-12

38. various, "*DII COE Real-Time ORB Trade Study*", Objective Interface Systems, Inc., February 2000.

39. "*DISA, Defense Information Infrastructure (DII) Common Operating Environment (COE) Baseline Specification*", version 3.1, April 29, 1997, DISA Joint Interoperability and Engineering Organization, Reston, Va.

40. Yang Jae-Heon, "*Scalable Synchronization in Shared Memory*", Doctor of Philosophy Dissertation, University of Maryland, 1994.

41. Nett Edgar, Mock Michael, "*A Recovery Model for Extended Real-Time Transactions*"**,** GMD – German National Research Center for Information Technology, Research Division SET-RS (Responsive Systems), Schloß Birlinghoven, D-53754 St. Augustin, Germany.

42. Engert Pamela, Surer Julie, **"***Introduction to the Defense Information Infrastructure(DII) Common Operating Environment (COE)",* CROSSTALK The Journal of Defense Software Engineering, September 1999, volume 12, number 9, pp 4-5.

43. Nett Edgar, Streich Hermann*, "Real-Time Scheduling of a Flexible Robot Application at Run-Time***",** German National Research Center for Information Technology (GMD), Research Division SET- RS (Responsive Systems), Schloss Birlinghoven, D-53754 St. Augustin, Germany, paper.

44. Nett Edgar, *"Real-Time Behaviour in a Heterogeneous Environment?",* IEEE WORDS'97, February 5-7, 1997, paper.

45. Nett Edgar, Gergeleit M., "*Preserving Real-Time Behavior In Dynamic Distributed Systems*", IASTED International Conference on Intelligent Information Systems 1997 (IIS'97), paper.

46. Nett Edgar, Gergeleit Martin, Streich Hermann, "*Flexible Resource Scheduling and Control in an Adaptive Real-Time Environment*", IASTED International Conference on Artificial Intelligence and Soft Computing, January 1997, paper.

47. Edgar Nett, Gergeleit M., Mock M., *"An Adaptive Approach to Object-Oriented Real-Time Computing"*, IEEE Proceedings of ISORC'98, 20-22 April 1998 in Kyoto, Japan, paper.

48. Gergeleit M., Kaiser H., Streich H., "*DIRECT: Towards a Distributed Object-Oriented Real-Time Control System*", German National Research Center for Information Technology, Research Division (Responsive Systems), Schloss Birlinghoven, D-53754 St. Augustin, Germany, paper.

49. Streich Hermann, *"TaskPair-Scheduling: An Approach for Dynamic Real-Time Systems"*, Proceedings 2nd Workshop on Parallel and Distributed Real-Time Systems, Cancun, Mexico, April 28-29, 1994.

50. Streich Hermann, Gergeleit M., "*On the Design of a Dynamic Distributed Real-Time Environment", German National Research Center for Information Technology, Research Division (Responsive Systems),Schloß Birlinghoven, D-53754 St. Augustin, Germany, paper.*

51. Moore Alan, "*Systems development – from conception to delivery", Embedded Systems Programming Europe, May 1999, pp 9-20.*

52. Luqi, Berzins V., Ge J., Shing M, Auguston M., Bryant R, Kin B., "*DCAPS – Architecture for Distributed Computer Aided Prototyping System*", Department of Computer Science , Naval Postgraduate School, 833 Dyer Road, Monterey, CA 93943 USA. Paper.

53. Department of Defense, *"Joint Technical Architecture"*, Version 3.1, 31 March 2000.

54. CICS application server platform, http://www-4.ibm.com/software/ts/cics.

55. French Ariane 5 disaster, http://www.ima.umn.edu/~arnold/disasters/ariane.html.

56. The Software Productivity Research consulting firm, http://www.spr.com/Resources/resources.htm.

57. The Waterfall Process, http://www.jsc.nasa.gov/bu2/PCEHHTML/pceh.htm.

58. Pilz, Markus, "*Earliest Deadline First Scheduling for Real-Time Java*", Embedded System Conference Europe 2000.

59. Cheesman J., Daniels J., "*UML Components, A Simple Process for Specifying Component-Based Software*", Addison-Wesley, October 2000.

60. Royce, W., "*Software Project Management*", Addison-Wesley 4th edition, June 1999.

61. Computer Networking, http://compnetworking.about.com/library/glossary/bldef-api.htm

62. Gurvine Jeff, Stauss Edwin, "Fundamentals of Tactical Missiles", Raytheon Missile Systems Company, 1998 – Limited Distribution.

63. Object Management Group, "OMG Unified Modeling Language Specification", version 1.4, September 2001.

64. Games, Richard, "Common Processors for Signal and Image Processing: Toward a Real-Time Embedded Common Operating Environment (RTE-COE)", MITRE Corporation, April 1998.

65. ISO/OSI Network Model, http://www.uwsg.iu.edu/usail/network/nfs/network_layers.html

66. P. Bernstein. "Middleware: A Model for Distributed System Services." *Communications of the AC*M, 39:2, February 1996, 86...98.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Professor Luqi
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

4. Professor Valdis Berzins
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

5. Professor ManTak Shing
   Naval Postgraduade School
   411 Dyer Road
   Monterey, CA 93943-5101

6. Dr. Nelson Ludlow
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

7. Professor Curt Schleher
   Naval Postygraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

8. Michael DaBose
   271 South Atlanta Drive
   Vail, AZ 85641-2315